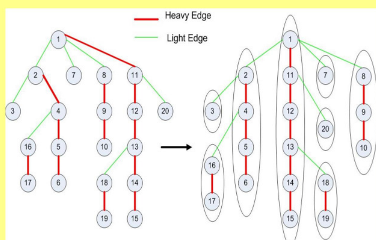
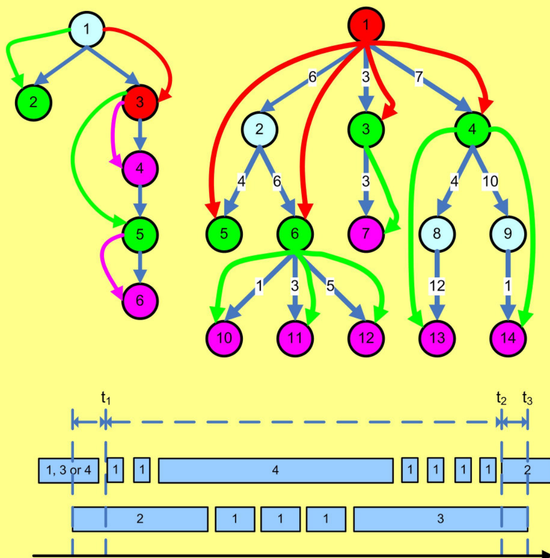
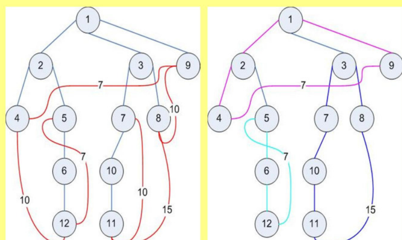
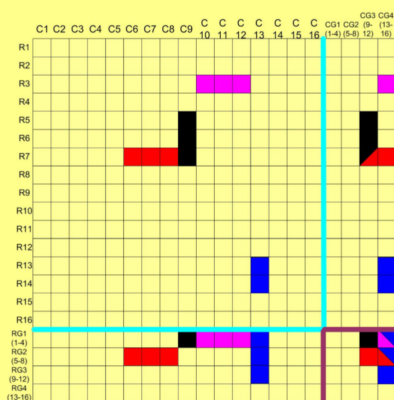
Mugurel Ionuț Andreica

Elemente de algoritmică – probleme şi soluții









CIBERNETICA MC

Bucureşti 2011

Mugurel Ionuţ Andreica
Elemente de algoritmică – probleme şi soluţii

Cuprins

Editura CIBERNETICA
Bucureşti 2011

ISBN: 978-606-8288-15-4

Introducere

Există multe cărţi care propun un număr mare de probleme spre rezolvare cititorilor, în timp ce prezintă doar soluţii ale unui număr mic dintre problemele propuse. Părerea proprie este că aceste cărţi nu îşi justifică existenţa decât parţial. Internet-ul este plin de enunţuri de probleme, de grade de dificultate variate (vezi doar câteva dintre link-urile de la sfârşitul acestei cărţi), majoritatea nefiind însoţite de explicaţii care să ajute la rezolvarea lor. În ziua de azi nu mai ducem lipsă de materiale de pregătire (existând atâtea surse de probleme online), ci de surse de explicaţii bine structurate şi comprehensibile.

Această carte a fost scrisă cu scopul de a suplini parţial lipsa surselor de explicaţii algoritmice structurate şi comprehensibile. Capitolele cărţii abordează tematici variate şi prezintă probleme având diverse grade de dificultate, însoţite de explicaţii complete şi structurate spre a fi uşor de lecturat şi înţeles.

Cartea conţine şi o arhivă de materiale auxiliare (enunţuri de probleme, programe sursă, descrieri de soluţii, etc.) care au scopul de a complementa explicaţiile structurate din cadrul cărţii.

Problemele discutate în această carte au fost propuse la concursuri şi olimpiade şcolare, precum olimpiadele naţionale din diverse ţări, olimpiade internaţionale, concursuri ACM ICPC, etc. Soluţiile prezentate sunt, în cea mai mare parte, elaborate în totalitate de către autorul cărţii. Un număr mic de descrieri de soluţii au fost preluate şi adaptate de la concursurile unde au fost propuse problemele respective. Problemele incluse în materialele auxiliare conţin cod sursă şi explicaţii scurte realizate de către autor, însă conţin, de asemenea, soluţiile oficiale (sau şi alte soluţii) ale problemelor respective.

Autorul este în prezent cadru didactic (şef de lucrări / lector) la Facultatea de Automatică şi Calculatoare, Universitatea Politehnica din Bucureşti. Conţinutul cărţii are la bază experienţa autorului de participant la multe concursuri naţionale şi internaţionale de informatică (olimpiada naţională şi internaţională de informatică, finala mondială a concursului ACM ICPC, etc.), precum şi de membru al comisiei ştiinţifice la astfel de concursuri (olimpiada naţională de informatică, olimpiada de informatică a Europei Centrale, olimpiada balcanică de informatică, concursul ACM ICPC – regiunea Europa de Sud-Est, concursuri .campion, etc.).

Capitolul 1. Programare Dinamică

Problema 1-1. 1expr (Happy Coding 2006, infoarena)

O *1-expresie* este definită în felul următor:

'1' sau ***'(' 1-expresie ')'*** sau **1-expresie '+' 1-expresie** sau
1-expresie '*' 1-expresie sau **1-expresie '^' 1-expresie** sau
1-expresie '!'

Deşi în cadrul expresiei apare în mod direct doar numarul *1*, rezultatele evaluării operaţiilor pot fi numere mai mari decât *1*. Pentru a evalua o expresie aritmetică, trebuie cunoscute priorităţile operatorilor. Operatorul cu cea mai mică prioritate este '+' şi realizează operaţia de adunare. Rezultatul 1-expresiei ***1+1+1*** este *3*. Operatorul '*' este mai prioritar decât operatorul '+' şi realizează operaţia de înmultire. Rezultatul 1-expresiei ***1+1*(1+1)*(1+1+1)+(1+1)*(1+1)*** este *1+1*2*3+2*2=1+6+4=11*. Operatorul '^' este mai prioritar decât operatorii '+' şi '*' şi realizează operaţia de ridicare la putere (*A^B* reprezintă *A* ridicat la puterea *B*). Rezultatul 1-expresiei ***(1+1)*(1+1+1)^(1+1)*(1+1+1)+(1+1)*** este *2*3^2*3+2=2*9*3+2=54+2=56*. Spre deosebire de operatorii '+' şi '*', care au proprietatea că *A+B=B+A* şi *A*B=B*A*, în cazul '^' nu este neapărat adevărat că *A^B=B^A*. O altă particularitate a acestui operator este ordinea de aplicare în cazul absenţei parantezelor: el este asociativ dreapta. De exemplu, *A^B^C^D* este echivalent cu *A^(B^(C^D))*. Rezultatul 1-expresiei ***(1+1)^(1+1)^(1+1+1)*** este *2^2^3=2^(2^3)=2^8=256* şi nu *(2^2)^3=4^3=64*. Astfel, dacă există mai mulţi operatori '^' neseparaţi de paranteze, ordinea de efectuare a operaţiilor este de la dreapta către stânga. Operatorul cu prioritatea cea mai mare este '!' şi realizează operaţia "factorial". Rezultatul 1-expresiei ***(1+1+1)!*** este *3!=6*. Factorialul unui numar *X*, notat *X!*, este definit ca *1*2*..*X*. Rezultatul 1-expresiei ***(1+1)*(1+1+1+1)!^(1+1+1)!!*** este *2*4!^3!!=2*(4!)^(3!!)=2*(4!)^((3!)!)=2*24^(6!)=2*(24^720)* (rezultatul este prea mare pentru a fi afisat aici).

Lungimea unei 1-expresii este egală cu numărul de caractere ale şirului ce reprezintă 1-expresia. Lungimea 1-expresiei ***(1+1)!^(1+1)*(1+1)+1*** este *20*. Dându-se o secvenţă de maxim *10.000* de valori, să se determine pentru fiecare număr *N* din secvenţă ($N \le 3^8$) o 1-expresie de lungime minimă al carei rezultat sa fie egal cu *N*.

Exemple:
N=1 => 1
N=3 => 1+1=1
N=200 => (1+1)^(1+1+1)*(1+(1+1+1+1)!)

Soluţie: Pentru fiecare *K* de la *1* la 3^8 se calculează lungimea minimă a unei 1-expresii care are ca rezultat pe *K*, iar ultima operatie efectuată în cadrul acestei 1-expresii este +, *, ^ sau ! (deci se calculează *4* valori). Este important să se calculeze toate aceste 4 valori şi nu doar o singură lungime minima a unei 1-expresii care il are ca rezultat pe *K*, deoarece este posibil ca acea 1-expresie să se obţină folosind operatori de prioritate mică si, pentru a fi folosită în cadrul unor expresii mai mari, să trebuiasca pusă între paranteze.

De exemplu, 1-expresia de lungimea minima pentru *8* este: ***1+1+(1+1+1)!***. O alta scriere a lui *8*, care are lungimea cu *1* mai mare este : ***(1+1)^(1+1+1)***. Însă această a doua reprezentare a lui *8* poate fi folosită fără a fi inclusă între paranteze, dacă trebuie înmulţită cu o altă expresie, în timp ce prima scriere trebuie inclusă între paranteze.

Relaţiile de recurenţă se determină destul de uşor (pentru un număr *K*):

- pentru '+' : se încearcă combinarea a două expresii al căror rezultat este *X*, respectiv *K-X*
- pentru '*' : combinarea a două expresii având valorile *X*, respectiv *K/X*
- pentru '^' : combinarea a *2* expresii având valorile *P* şi *Q* (unde $P^Q=K$)
- pentru '!' : numai în cazul în care K este factorialul unui numar (pentru limitele date, K poate fi maxim 7!)

Se calculează cele *4* valori pentru fiecare număr de la 1 la limita maximă la inceput, apoi pentru fiecare număr din secvenţă doar se afişază rezultatul (dacă s-ar recalcula valorile pentru fiecare număr, s-ar depasi limita de timp).

Pentru determinarea relaţiilor de recurenţă vom considera cazul mai general, în care avem *Q* operaţii binare, *op(1)*, ..., *op(Q)*, în ordinea crescătoare a priorităţilor lor (de ex., *op(1)=+, ..., op(Q)=^*). Vom calcula *Lmin(i,j)*=lungimea minimă a unei expresii al cărei rezultat este *i*, astfel încât ultima operaţie efectuată să fie *op(j)*. După calcularea acestor valori pentru un *i*, vom calcula valorile *LRmin(i,j)* şi *RLmin(i,j)*:

- $LRmin(i,0)=RLmin(Q+1)=+\infty$;
- $LRmin(i,1\leq j\leq Q)=min\{LRmin(i,j-1), Lmin(i,j)\}$; $RLmin(i,j)=min\{RLmin(i,j+1), Lmin(j)\}$.

Pentru a calcula o valoare *Lmin(i,j)* vom determina cele două numere *X* şi *Y*, astfel încât *X op(j) Y = i*. Dacă operaţia *op(j)* este comutativă, atunci $Lmin(i,j)=min\{RLmin(X,j), RLmin(X,1)+2\}+min\{RLmin(Y,j), RLmin(Y,1)+2\}$.

Dacă *op(j)* nu este comutativă, dar este asociativă dreapta (cum este operaţia de ridicare la putere), atunci $Lmin(i,j)=min\{RLmin(X,j+1), RLmin(X,1)+2\}+min\{RLmin(Y,j), RLmin(Y,1)+2\}$.

Dacă *op(j)* este necomutativă şi asociativă stânga, atunci $Lmin(i,j)=min\{RLmin(X,j), RLmin(X,1)+1\}+min\{RLmin(Y,j+1), RLmin(Y,1)+2\}$.

Problema 1-2. Pali (Happy Coding 2007, infoarena)

Orice cuvânt se poate împărţi într-un număr mai mare sau mai mic de subsecvenţe, fiecare subsecvenţă fiind un palindrom (în cel mai rău caz, fiecare subsecvenţă are lungimea *1*). Fiind dat un cuvânt, determinaţi numărul minim de palindroame în care poate fi împărţit acesta.

Exemple:

aaaabbaa => *2* palindroame: **aa + aabbaa**
abccbazzz => *2* palindroame: **abccba + zzz**

Soluţie: Se va calcula *pmin[i]*=numărul minim de palindroame în care poate fi împărţit şirul format din primele *i* caractere ale sirului dat.

Soluţia 1

Vom avea *pmin[0]=0* şi pmin[i]=1+min{pmin[j]|0≤j≤i-1 şi subşirul format din caracterele de pe poziţiile *j+1,..,i* este palindrom }. Pentru a determina rapid dacă subşirul dintre *2* poziţii *k* şi *l* este un palindrom, vom calcula o matrice *Pali[k][l]*, având valorile *1* şi *0*, dacă subşirul dintre poziţiile *k* şi *l* este palindrom sau nu. Avem *Pali[i][i]=1* şi *Pali[i][i+1]=1*, dacă *sir[i]=sir[i+1]* (*0*, altfel).

Pentru $l-k\geq 2$, *Pali[k][l]=1*, dacă *sir[k]= sir[l]* şi *Pali[k+1][l-1]=1*.

Soluţia 2

Vom încerca să calculăm valorile *pmin[i]* treptat. Vom parcurge poziţiile de la *1* la *N* (lungimea şirului) şi vom incerca să începem un palindrom având centrul în poziţia *i* (pentru cazul unui palindrom de lungime impară), respectiv având centrul la poziţiile *i* şi *i+1* (pentru

cazul unui palindrom de lungime pară). Ideea importantă este că, atunci când ajungem la poziţia *i*, toate valorile *pmin[j]* ($j<i$) au fost calculate, iar valorile *pmin[k]* ($k>i$), au nişte valori parţiale (adică valorile calculate nu sunt încă finale, ele mai putând să scadă ulterior).

Vom extinde treptat cât putem de mult un palindrom în jurul poziţiei *i* (respectiv poziţiilor *i* şi *i+1*); la fiecare pas de extindere, vom incrementa palindromul de la lungimea curentă *L*, la lungimea *L+2* (pentru aceasta verificăm caracterele de la capatele palindromului); înainte de a realiza extinderea, vom incerca să folosim palindromul curent în cadrul unei solutii - mai exact, avem următoarele posibilităţi: *pmin[i+(L+1)/2-1]=min{pmin[i+(L+1)/2-1], pmin[i-(L+1)/2]+1}* (pentru cazul lungimii impare), respectiv *pmin[i+L/2]= min{pmin[i+L/2], pmin[i-L/2]+1}* poate fi folosită pentru a micşora valoarea lui *pmin[i+L/2]* (pentru cazul lungimii pare).

Complexitatea ambelor solutii este $O(N^2)$, însă a doua soluţie merge mai repede, în practică, deoarece ia în considerare numai palindroamele valide (care pot fi cel mult $O(N^2)$).

Problema 1-3. Bitmap (Happy Coding 2007, infoarena)

Nişte cercetători au inventat un mod de a codifica un bitmap de dimensiuni *MxN* ($1\leq M,N\leq 11$) sub forma unui şir de caractere. Şirul este, de fapt, o reprezentare a unor operaţii efectuate asupra bitmap-ului. Algoritmul de codificare este descris în continuare:

- dacă toate celulele bitmap-ului au culoarea *1*, atunci codificarea este *"1"*
- dacă toate celulele bitmap-ului au culoarea *0*, atunci codificarea este *"0"*
- altfel, trebuie să alegeti una din următoarele modalităţi pentru a codifica bitmap-ul:
    - *Codificare(B) = "A" + Codificare(B_1) + Codificare(B_2)*, unde *"+"* reprezintă operaţia de concatenare, B_1 este bitmap-ul rezultat prin păstrarea primelor *M/2* linii ale bitmap-ului *B* şi a tuturor coloanelor, iar B_2 este bitmap-ul rezultat prin păstrarea ultimelor *M-(M/2)* linii ale bitmap-ului *B* şi a tuturor coloanelor (astfel, B_1 va avea *M/2* linii şi *N* coloane, iar B_2 va avea *M-(M/2)* linii şi *N* coloane).
    - *Codificare(B) = "B" + Codificare(B_1) + Codificare(B_2)*, unde B_1 este bitmap-ul rezultat prin păstrarea primelor *N/2* coloane ale bitmap-ului *B* şi a tuturor liniilor, iar B_2 este bitmap-ul rezultat prin păstrarea ultimelor *N-(N/2)* linii ale bitmap-ului *B* şi a tuturor liniilor (astfel, B_1 va avea *M* linii şi *N/2* coloane, iar B_2 va avea *M* linii şi *N-(N/2)* coloane).
    - *Codificare(B) = "C" + Codificare(B_1) + Codificare(B_2)*, unde B_1 este bitmap-ul rezultat prin păstrarea tuturor liniilor cu numere impare ale bitmap-ului *B* (liniile sunt numerotate începând de la *1*) şi a tuturor coloanelor, iar B_2 este bitmap-ul rezultat prin păstrarea tuturor liniilor cu numere pare ale bitmap-ului *B* şi a tuturor coloanelor (astfel, B_1 va avea *M-(M/2)* linii şi *N* coloane, iar B_2 va avea *M/2* linii şi *N* coloane).
    - *Codificare(B) = "D" + Codificare(B_1) + Codificare(B_2)*, unde B_1 este bitmap-ul rezultat prin păstrarea tuturor coloanelor cu numere impare ale bitmap-ului *B* (coloanele sunt numerotate începând de la *1*) şi a tuturor liniilor, iar B_2 este bitmap-ul rezultat prin păstrarea tuturor coloanelor cu numere pare ale bitmap-ului *B* şi a tuturor liniilor (astfel, B_1 va avea *M* linii şi *N-(N/2)* coloane, iar B_2 va avea *M* linii şi *N/2* coloane).

Fiind dat *T* ($1\leq T\leq 1500$) teste (bitmap-uri), gasiti lungimea celei mai scurte codificări pentru fiecare bitmap.

Soluţie: O primă soluţie ar consta în a încerca toate cele *4* posibilităţi de a codifica un bitmap care nu are toate celulele de aceeaşi culoare. Această abordare de tip backtracking nu s-ar incadra in limita de timp. Optimizarea necesară constă în memoizarea stărilor prin care trecem în cadrul backtracking-ului (algoritmul nu se mai numeste backtracking acum, ci devine un fel de programare dinamica). O stare este dată de *2* numere pe cel mult *11* biţi, S_1 şi S_2. Biţii de *1* din S_1 definesc liniile selectate din cadrul bitmap-ului initial, iar biţii de *1* din S_2 definesc coloanele selectate din cadrul bitmap-ului iniţial. Bitmap-ul definit de starea (S_1,S_2) este reprezentat de celulele aflate la intersecţia dintre liniile selectate de S_1 şi coloanele selectate de S_2. Prin urmare, vom calcula $lmin[S_1,S_2]$=lungimea minimă a codificării bitmap-ului ce corespunde stărilor S_1 şi S_2.

$lmin[2^M-1,2^N-1]$ va conţine rezultatul căutat. Pentru ca soluţia să se încadreze în timp, matricea cu valorile asociate fiecărei stări nu trebuie reiniţializată la fiecare test. Se va folosi o matrice suplimentară *last_test*, în care vom memora numărul ultimului test în care am ajuns să calculăm valoarea pentru starea (S_1,S_2). Dacă ajungem la starea (S_1,S_2) în testul curent, vom verifica valoarea lui $last_test[S_1,S_2]$. Dacă aceasta este egală cu numărul testului curent, înseamnă că am calculat deja valoarea respectivă; în caz contrar, o vom calcula acum şi vom seta $last_test[S_1,S_2]$ la numărul testului curent.

Problema 1-4. Palin (Olimpiada Internaţională de Informatică 2000, enunţ modificat)

Se dă un şir format din *N* ($1 \le N \le 5000$) de caractere. Pentru fiecare caracter al alfabetului *c*, se dă un cost *cost(c)*. Determinaţi costul total minim al caracterelor ce trebuie inserate în şir, pentru ca acesta să devină palindrom.

Exemplu:

N=5; sirul=Ab3bd => răspunsul este *2* (se obţine şirul ***A**d**b3bd**A*)

Soluţie: Vom calcula următoarea matrice: *Cmin[i,j]*=costul total minim al caracterelor ce trebuie inserate pentru ca subşirul format din caracterele de pe poziţile *i, i+1, ..., j* ($i \le j$) să devină palindrom.

Avem *Cmin[i,i]=0. Cmin[i,i+1]=0*, dacă *şir[i]=şir[i+1]* (*şir[p]*=al *p*-lea caracter al şirului dat); altfel, *Cmin[i,i+1]=1*. Pentru $j>i+1$, avem următoarele relaţii:

- dacă *şir[i]=şir[j]*, atunci *Cmin[i][j]=Cmin[i+1][j-1]*
- altfel, *Cmin[i][j]=min{cost(şir[i])+Cmin[i+1][j], cost(şir[j])+Cmin[i][j-1]}*

Răspunsul îl avem în *Cmin[1][N]*. Valorile *Cmin[i][j]* se calculează în ordinea lungimii intervalului *[i,j]*. Observăm că, atunci când calculăm perechile *(i,j)* cu $j-i+1=k$, avem nevoie doar de valorile pentru perechi *(i',j')*, cu $j'-i'+1=k-1$. Astfel, în orice moment în cadrul execuţiei algoritmului, avem nevoie doar de *O(N)* valori în memorie.

O altă metodă de rezolvare constă în a calcula *MinC[i][j]*=costul total minim al caracterelor ce trebuie inserate pentru ca subşirul format din primele *i* caractere ale şirului să fie transformat în subşirul format din ultimele *j* caractere ale şirului. Avem $MinC[0][j \ge 0]=0$ şi $MinC[i \ge 0][0]=i$. Pentru $i \ge 1$ şi $j \ge 1$, avem:

- dacă *şir[i]=şir[N-j+1]*, atunci *MinC[i][j]=MinC[i-1][j-1]*
- altfel, *MinC[i][j]=min{cost(şir[N-j+1])+MinC[i][j-1], cost(şir[i])+MinC[i-1][j]}*

Rezultatul este $min\{min\{MinC[i][i+1] | 0 \le i \le N\}, min\{MinC[i][i+2] | 0 \le i \le N-1\}\}$. Şi în acest caz putem folosi doar *O(N)* memorie.

Problema 1-5. Roboţi (Olimpiada Baltică de Informatică, 2002, enunţ modificat)

Se dau *N* (*1≤N≤10*) roboţi. Fiecare robot *i* (*1≤i≤N*) este amplasat la coordonatele *(x(i),y(i))*, se uită în direcţia *d(i)* (*Nord*, *Est*, *Sud* sau *Vest*) şi are propriul său program *P(i)*. Un program *P(i)* constă dintr-o secvenţă de *nc(i)* (*0≤nc(i)≤ 50*) comenzi. Vom nota prin *P(i,j)* a *j*-a comandă din programul *P(i)*. Comenzile sunt de două tipuri:

(1) rotaţie cu 90 de grade spre stânga/spre dreapta ;

(2) deplasare cu o poziţie în direcţia în care se uită.

Dacă robotul se află la coordonatele *(xr,yr)* şi are de executat o comandă de deplasare, atunci: dacă se uită în direcţia *Nord/Est/Sut/Vest*, noile sale coordonate sunt: *(xr,yr-1)/(xr+1,yr)/(xr,yr+1)/(xr-1,yr)*. Fiecare comandă *P(i,j)* are un cost de ştergere *C(i,j)*. Se doreşte ştergerea unor comenzi din programele roboţilor astfe încât, în urma comenzilor efectuate, toţi roboţii să ajungă în aceeaşi poziţie. Costul total al comenzilor şterse trebuie să fie minim.

Soluţie: Pentru fiecare robot *i* vom determina mulţimea de poziţii *(xr(i),yr(i),dr(i))* în care poate ajunge robotul în urma efectuării comenzilor din programul său, în cazul în care unele comenzi pot fi şterse, împreună cu costul minim pentru a ajunge la poziţiile respective (*dr(i)* este direcţia în care se uită robotul). Vom nota prin *S(i,j)*=mulţimea stărilor în care poate ajunge robotul *i* dacă luăm în considerare doar comenzile *1, ..., j*.

Avem *S(i,0)={poziţie=(x(i),y(i),d(i)), cost=0}*. Pentru *1≤j≤nc(i)*, iniţializăm *S(i,j)* cu *S(i,j-1)* (cazul în care comanda *P(i,j)* nu se execută). Apoi considerăm fiecare poziţie din *S(i,j-1)*, la care efectuăm comanda *P(i,j)*. Să presupunem că, în urma considerării poziţiei *(xr, yr, dr)* cu costul *cr* şi a efectuării comenzii *P(i,j)*, obţinem poziţia *(xr', yr', dr')* cu costul *cr'*. Dacă poziţia *(xr', yr', dr')* nu se află în *S(i,j)*, o introducem, setând costul *cr'*. Dacă poziţia se află în *S(i,j)* cu un cost mai mare, atunci modificăm costul la *cr'*.

Putem implementa mulţimile *S(i,j)* ca nişte arbori echilibraţi sau, deoarece fiecare robot se poate plimba doar în poziţiile la distanţă cel mult *nc(i)* faţă de poziţia iniţială, putem menţine un vector caracteristic pentru fiecare tuplu *(xdr, ydr, dr)* (unde *xdr* şi *ydr* sunt între *–nc(i)* şi *+nc(i)*, *dr* este între *1* şi *4*, poziţia reală fiind *(x(i)+xdr, y(i)+ydr, dr)*).

La final, vom determina mulţimea *SS(i)*, conţinând toate poziţiile *(xr,yr)* unde poate ajunge robotul, indiferent de orientare, împreună cu costul acestora (practic, pentru fiecare poziţie *(xr,yr,dr)* cu costul *cr*, introducem *(xr,yr)* cu costul *cr* în *SS(i)*, dacă *(xr,yr)* nu există acolo, sau micşorăm costul asociat lui *(xr,yr)* în *SS(i)* la *cr*, dacă avea un cost asociat mai mare decât *cr*).

În continuare, vom considera fiecare poziţie *(xr, yr)* din *SS(1)*. Pentru fiecare poziţie, vom calcula costul total pentru ca toţi roboţii să ajungă în poziţia respectivă. Vom parcurge toţi roboţii *j* (*1≤j≤N*). Dacă poziţia *(xr,yr)* nu există în *SS(j)*, atunci costul total corespunzător ei va fi +∞. Dacă poziţia *(xr,yr)* apare în toate mulţimile *SS(j)* (*1≤j≤N*), atunci costul ei total va fi suma costurilor asociate în fiecare mulţime. Răspunsul problemei este dat de poziţia *(xr,yr)* pentru care costul total calculat este minim.

Problema 1-6. Drum de valoare maximă într-un graf orientat aciclic

Se dă un graf orientat aciclic cu *N* noduri. Fiecare nod *i* (*1≤i≤N*) are o valoare *v(i)*. Dorim să determinăm un drum în acest graf, pentru care suma valorilor este maximă. În plus, pentru fiecare nod *i* se cunoaşte o submulţime de noduri *S(i)*. În cadrul drumului, valoarea unui nod

i poate fi înlocuită cu valoarea oricărui alt nod din mulţimea $S(i)$. Totuşi, se pot realiza maxim K ($0 \le K \le N$) astfel de înlocuiri.

Soluţie: Vom calcula $Vmax(i,j)$=suma maximă a unui drum ce se termină la nodul i şi în care s-au efectuat j înlocuiri ($0 \le j \le K$). Pentru început, vom calcula pentru fiecare nod valoarea $vs(i)=max\{v(j)|j \in S(i)\}$. Vom calcula apoi o sortare topologică a nodurilor: $o(1), ..., o(N)$ şi vom parcurge nodurile în ordinea din cadrul sortării topologice ($i=1,...,N$).

Iniţializăm $Vmax(o(i), j)=max\{Vmax(x, j)+v(j)|$ există muchia orientată $(x,o(i))$ în graf$\}$ ($0 \le j \le K$). Apoi considerăm toate valorile $1 \le j \le K$ şi setăm $Vmax(o(i),j)=max\{Vmax(o(i),j), max\{Vmax(x,j-1)+vs(j)|$ există muchia orientată $(x,o(i))$ în graf $\}\}$.

Drumul de valoare maximă este $max\{Vmax(i,j)|1 \le i \le N, 0 \le j \le K\}$. Observăm că $max\{Vmax(i,0)|1 \le i \le N\}$ este drumul de valoare maximă fără nicio interschimbare. Complexitatea algoritmului este $O((N+M) \cdot (K+1))$.

Variante ale acestei probleme au fost propuse la multe concursuri de programare (de ex., Olimpiada Baltică de Informatică, 2000).

Problema 1-7. Traversarea podului cu o lanternă

De o parte a unui pod se află N ($1 \le N \le 100.000$) persoane. Fiecare persoană i are un timp $T(i)$ de traversare a podului. Afară este noapte, există o singură lanternă şi se doreşte traversarea podului de către toate cele N persoane. Pe pod pot trece simultan maxim două persoane, dar, întrucât există o singură lanternă, ele vor circula la viteza persoanei mai lente (durata de traversare simultană a podului a persoanelor i şi j este $max\{T(i),T(j)\}$). Determinaţi o strategie de traversare a podului cu durată totală minimă.

Soluţie: Vom sorta persoanele astfel încât să avem $T(1) \le T(2) \le ... \le T(N)$. Vom calcula $Tmin(i)$=timpul minim necesar pentru traversarea podului de către primele i persoane (în ordinea sortată după timpii de traversare).

Avem $Tmin(1)=T(1)$, $Tmin(2)=max\{T(1), T(2)\}=T(2)$, $Tmin(3)=T(2)+T(1)+T(3)$. Pentru $i \ge 4$ avem $Tmin(i)=min\{Tmin(i-1)+T(1)+T(i), Tmin(i-2)+T(1)+T(i)+ T(2)+T(2)\}$. Prima variantă corespunde următorului caz: au traversat podul primele $i-1$ persoane, persoana 1 se întoarce cu lanterna şi trece podul împreună cu persoana i. A doua variantă corespunde următorului caz: au traversat podul primele $i-2$ persoane; se întoarce pe partea cealaltă persoana 1, împreună cu lanterna; trec podul persoanele $i-1$ şi i împreuna; se întoarce pe partea cealaltă persoana 2, cu lanterna; trec podul persoanele 1 şi 2 împreună.

Complexitatea algoritmului este $O(N \cdot log(N))$ (faza de sortare), plus $O(N)$ (faza de programare dinamică).

Problema 1-8. Cuplaj de Cost Maxim într-un Arbore (Olimpiada de Informatică a Europei Centrale, 2007, enunţ modificat)

Se dă un arbore cu N ($1 \le N \le 5000$) noduri. Fiecare muchie (u,v) are un cost $c(u,v) \ge 0$. Determinaţi un cuplaj de cost maxim, precum şi numărul de cuplaje de cost maxim. Un cuplaj este o submulţime de muchii astfel încât oricare două muchii din cuplaj nu au niciun capăt comun. Costul unui cuplaj este egal cu suma costurilor muchiilor din cuplaj.

Soluţie: Vom alege o rădăcină pentru arbore (un nod oarecare r), definind astfel relaţiile de părinte, fiu, subarbore, etc. Vom calcula, pentru fiecare nod i, câte patru valori: $CA(i)$=costul maxim al unui cuplaj considerând doar nodurile din subarborele lui i şi care conţine o muchie

adiacentă cu nodul i; $CB(i)$=o valoare având aceeaşi semnificaţie ca şi $CA(i)$, doar că nu există nicio muchie în cuplaj care să fie adiacentă cu nodul i; $NCA(i)$=numărul de cuplaje de cost maxim în subarborele nodului i ce conţin o muchie adiacentă cu nodul i; $NCB(i)$= numărul de cuplaje de cost maxim în subarborele nodului i ce nu conţin o muchie adiacentă cu nodul i.

Vom parcurge arborele de la frunze spre rădăcină. Pentru o frunză i avem $CA(i)=-\infty$, $CB(i)=0$ şi $NC(i)=1$. Pentru un nod neterminal i vom calcula $CB(i)$ ca fiind suma valorilor $max\{CA(j), CB(j)\}$, cu j fiu al lui i. Vom determina, de asemenea, acel fiu k al lui i pentru care valoarea ($max\{CA(k), CB(k)\}-CB(k)$) este minimă între toţi ceilalţi fii.

$CA(i)$ va fi egal cu $c(i,k)+CB(i)+CB(k)-max\{CA(k),CB(k)\}$ (muchia adăugată la cuplajul de cost maxim este muchia i-k).

$NCB(i)$ este calculat ca produs al următoarelor valori: $NCmax(j)$=(dacă $CA(j)>CB(j)$ atunci $NCA(j)$ altfel dacă $CA(j)<CB(j)$ atunci $NCB(j)$ altfel ($NCA(j)+NCB(j)$)), unde j ia ca valori, pe rând, fiecare fiu al lui i.

Pentru a calcula $NCA(i)$ îl iniţializăm la 0 şi apoi considerăm, pe rând, fiecare fiu j al lui i. Dacă $CA(i)=c(i,j)+CB(i)+CB(j)-max\{CA(j), CB(j)\}$, atunci setăm $NCA(i)= NCA(i)+NCB(i)/NCmax(j)\cdot NCB(j)$, unde $NCmax(j)$ a fost definită anterior.

Costul maxim al unui cuplaj este $max\{CA(r), CB(r)\}$, iar numărul de cuplaje maxime este egal cu $NCmax(r)$ (folosind definiţia dată mai sus). Complexitatea algoritmului este $O(N)\cdot NrMari(N)$, unde $NrMari(N)$ este complexitatea lucrului (adunări, scăderi, înmulţiri, împărţiri) cu numere mari având $O(N)$ cifre.

Problema 1-9. Ţestoase (ACM ICPC NEERC, 2004)

N ($1\le N\le 50.000$) ţestoase se mişcă pe o stradă cu aceeaşi viteză. Fiecare ţestoasă vede ţestoasele dinaintea ei şi de după ea. Unele ţestoase se mişcă în grup, astfel că o ţestoasă nu poate vedea celelalte ţestoase din acelaşi grup (nu sunt înainte sau după ea, ci chiar în dreptul ei). Fiecare ţestoasă i ($1\le i\le N$) spune că sunt exact $a(i)$ ţestoase înaintea ei şi exact $b(i)$ ţestoase după ea (i este un simplu identificator al unei ţestoase care nu are legătură cu ordinea ţestoaselor pe stradă). Determinaţi numărul minim de ţestoase care mint.

Exemplu:

N=5 **a(1)=0 ; b(1)=2** **a(2)=0 ; b(2)=3** **a(3)=2 ; b(3)=1** **a(4)=1 ; b(4)=2** **a(5)=4 ; b(5)=0**	**Numărul minim de ţestoase care mint este 2: ţestoasele 1 şi 4.**

Soluţie: Să considerăm că fiecare ţestoasă i are asociată o coordonată $x(i)$ pe stradă. Vom considera că o ţestoasă j se află înaintea (după/în dreptul) unei ţestoase i dacă $x(j)>x(i)$ ($x(j)<x(i)$ / $x(j)=x(i)$). Vom considera ţestoasele sortate descrescător după coordonata x asociată $(o(1), ..., o(N))$, $x(o(i))\ge x(o(i+1))$, $1\le i\le N-1$ (această ordonare nu este cunoscută).

Dacă mai multe ţestoase se află la aceeaşi coordonată x, atunci ele se pot afla în orice ordine în cadrul ordonării. Fiecare afirmaţie a unei ţestoase i ($1\le i\le N$) este echivalentă cu a spune că ţestoasele $o(j)$ cu j în intervalul de poziţii $[front(i)=a(i)+1, back(i)=N-b(i)]$ din cadrul ordonării o sunt la coordonate x egale, toate ţestoasele $o(j)$ cu j în intervalul de poziţii $[1,a(i)]$ din cadrul ordonării o sunt la coordonate x mai mari, şi toate ţestoasele $o(j)$ cu j în intervalul de poziţii $[N-b(i)+1,N]$ din cadrul ordonării o sunt la coordonate x mai mici.

Vom ignora intervalele cu *back(i)<front(i)* (ce corespund unor ţestoase ce mint în mod obligatoriu) şi vom sorta apoi celelalte intervale *[front(i), back(i)]* crescător după capătul stânga (şi, pentru acelaşi capăt stânga, crescător după capătul dreapta). Apoi vom păstra doar cele $M \le N$ intervale distincte şi vom asocia o pondere fiecărui interval păstrat, egală cu numărul de apariţii ale intervalului în cadrul sortării; dacă ponderea intervalului este mai mare decât lungimea sa (lungimea unui interval *[p,q]* este $q-p+1$), atunci ponderea este setată la lungimea acestuia.

În acest moment avem următoarea problemă. Dându-se $M=O(N)$ intervale *[left(i), right(i)]*, fiecare având o pondere $w(i)$ ($1 \le i \le M$), determinaţi o mulţime de intervale cu proprietatea că oricare două intervale nu se intersectează, a căror sumă a ponderilor este maximă. Suma ponderilor acestor intervale corespunde unui număr maxim de ţestoase care spun adevărul. Celelalte ţestoase vor fi distribuite în intervalele din mulţime a căror pondere este mai mică decât lungimea lor, iar ţestoasele rămase după această redistribuire vor forma câte un grup singure, pe acele poziţii (de la *1* la *N*) care nu aparţin niciunui interval din mulţimea selectată.

Pentru a rezolva problema la care am ajuns vom sorta crescător cele $2 \cdot M$ capete de intervale, reţinând, pentru fiecare capăt *p*, tipul său *tip(p)* (*stânga* sau *dreapta*) şi indicele intervalului din care face parte, *idx(p)*. Dacă avem mai multe capete de interval cu aceeaşi valoare, atunci capetele stânga se vor afla înaintea capetelor dreapta cu valoarea respectivă.

Vom parcurge apoi şirul celor $2 \cdot M$ capete de interval şi pentru fiecare poziţie *p* vom calcula o valoare *Wmax(p)*=suma maximă a ponderilor unei submulţimi de intervale care nu se intersectează, ale căror capete dreapta se află pe poziţii $q \le p$. Vom iniţializa *Wmax(0)=0*. Dacă *tip(p)=stânga*, atunci setăm *pozleft(idx(p))=p* şi *Wmax(p)=Wmax(p-1)*. Dacă *tip(p)=dreapta*, atunci *Wmax(p)=max{Wmax(p-1), w(p)+Wmax(pozleft(idx(p))-1)}*; primul caz corespunde situaţiei în care intervalul *idx(p)* nu este selectat, iar al doilea caz corespunde situaţiei în care intervalul *idx(p)* este selectat.

$Wmax(2 \cdot M)$ este suma maximă a ponderilor din mulţimea căutată. Mulţimea poate fi determinată dacă urmărim modul în care au fost calculate valorile *Wmax(*)*. Aşadar, această problemă a fost rezolvată într-o complexitate $O(M \cdot log(M))$, obţinând o soluţie de complexitate $O(N \cdot log(N))$ pentru problema iniţială.

Problema 1-10. Operaţii fiscale (TIMUS)

Se dau trei numere *A*, *B* şi *C* de cel mult *1.000* de cifre (în baza *Q*, $2 \le Q \le 100$). Modificaţi cifrele numerelor *A*, *B* şi *C*, în aşa fel încât suma lor să fie numărul *C*. Dacă cifra *i* unui număr se modifică de la valoarea *x* la *y*, se plăteşte un cost egal cu $|x-y|$. Efectuaţi modificările astfel încât costul total plătit să fie minim.

Soluţie: Vom extinde cele *3* numere astfel încât să aibă toate acelaşi număr *N* de cifre (adăugând *0*-uri la început). Vom nota prin *X(i)* a *i*-a cifră a numărului *X* (*X=A, B* sau *C*; cifrele se numără de la cea mai puţin semnificativă la cea mai semnificativă). Vom calcula *CA(i)*=costul total minim plătit pentru ca adunând numerele formate din cifrele de pe poziţiile *1,...,i* ale numerelor *A* şi *B* să obţinem cifrele de pe poziţiile *1,...,i* ale numărului *C* şi să nu avem transport; *CB(i)*=aceeaşi semnificaţie, dar avem transport.

Avem *CA(0)=0* şi $CB(0)=+\infty$. Pentru o poziţie *i* vom iniţializa $CA(i)=CB(i)=+\infty$, apoi vom încerca toate posibilităţile *cx* pentru cifra *i* din numărul *A* şi *cy* pentru cifra *i* din numărul *B* ($0 \le cx,cy \le Q-1$; avem restricţii suplimentare în cazul în care poziţia *i* este o cifră

extinsă a numărului, caz în care ea trebuie să rămână 0, sau dacă este cea mai semnificativă cifră a numărului dinainte de extindere, caz în care $cx,cy \geq 1$).

Pentru fiecare posibilitate calculăm $sxy=cx+cy$ şi considerăm două cazuri: *(1)* nu avem transport de la poziţia anterioară => calculăm $cxy=sxy \text{ div } Q$ şi $rxy=sxy \text{ mod } Q$; setăm $costxy=|cx-A(i)|+|cy-B(i)|+|rxy-C(i)|+CA(i-1)$; *(2)* avem transport de la poziţia anterioară => calculăm $cxy=(sxy+1) \text{ div } Q$ şi $rxy=(sxy+1) \text{ mod } Q$; setăm $costxy=|cx-A(i)|+|cy-B(i)|+|rxy-C(i)|+CB(i-1)$. Apoi, dacă $cxy>0$, setăm $CA(i)=min\{CA(i), costxy\}$; altfel, setăm $CB(i)=min\{CB(i), costxy\}$. Complexitatea soluţiei este $O(N \cdot Q^2)$.

Putem reduce complexitatea la $O(N \cdot Q)$, după cum urmează. În loc să încercăm toate combinaţiile de posibilităţi cx şi cy la un pas i, vom considera toate posibilităţile de sume sxy (de la 0 la $2 \cdot Q-2$). Dacă $sxy \geq A(i)+B(i)$, vom seta $cx=A(i)+min\{sxy-(A(i)+B(i)), B-1-A(i)\}$ şi $cy=sxy-cx$. Dacă $sxy<A(i)+B(i)$, atunci setăm $cx=A(i)-min\{A(i)+B(i)-sxy, A(i)\}$ şi $cy=sxy-cx$.

O situaţie interesantă apare atunci când trebuie să plătim un cost $wA \geq 0$ pentru modificarea cu o unitate a unei cifre din A, $wB \geq 0$ pentru modificarea cu 1 a unei cifre din B, respectiv $wC \geq 0$ pentru modificarea cu 1 a unei cifre din C. Pentru ambii algoritmi descrişi (de complexitate $O(N \cdot Q^2)$ şi $O(N \cdot Q)$) vom calcula $costxy$ ca fiind: $wA \cdot |cx-A(i)|+wB \cdot |cy-B(i)|+wC \cdot |rxy-C(i)|+CA(i-1)$, respectiv ca $wA \cdot |cx-A(i)|+wB \cdot |cy-B(i)|+wC \cdot |rxy-C(i)|+CB(i-1)$. Pentru al doilea algoritm (de complexitate $O(N \cdot Q)$), o dată ce fixăm suma sxy, dacă $wA \leq wB$, atunci vom folosi procedeul descris mai sus. Dacă $wB<wA$, atunci setăm întâi cy şi apoi calculăm cx ca $sxy-cy$. Dacă $sxy \geq A(i)+B(i)$, cy va fi egal cu $B(i)+min\{sxy-(A(i)+B(i)), Q-1-B(i)\}$; altfel, $cy=B(i)-min\{A(i)+B(i)-sxy, B(i)\}$.

Problema 1-11. Alegeri (Lotul Naţional de Informatică, România 2007)

În ţara Apulumia se apropie timpul alegerilor pentru preşedinte. În ţară sunt un număr de N ($3 \leq n \leq 2.000.000.000$) de alegători. Din aceştia, doar o mică parte îl susţin în continuare pe actualul preşedinte, care, în mod natural, doreşte să fie reales.

Pentru o „desfăşurare democratică" a alegerilor se aplică urmatoarea procedură: toţi alegătorii sunt împărţiţi în grupe egale, care cu majoritate simplă (jumătate plus unu) aleg un reprezentant, apoi reprezentanţii sunt împărţiţi în grupe egale, care la rândul lor aleg un reprezentant, şi asa mai departe, după acelaşi principiu, până la desemnarea unui singur reprezentant, preşedintele. Actualul preşedinte împarte alegătorii în grupe şi plasează susţinătorii după bunul său plac. Ajutaţi-l să determine numărul minim de susţinători ai săi, care plasaţi corespunzător duc la câştigarea alegerilor. Într-o grupă, la egalitate de voturi, câştigă opoziţia.

Exemplu: N=9 => Sunt necesari 4 susţinători.

Soluţie: Se determină toţi divizorii lui N, şi apoi se sortează în ordine crescătoare (în vectorul dv, unde $dv[k]$ reprezintă al k-lea divizor al lui N; $1 \leq k \leq NrDivizori$). Apoi se calculează în vectorul $nmin[i]$ numărul minim de susţinători necesari pentru a câştiga alegerile, dacă ar exista $dv[i]$ alegători în total.

Evident, $nmin[1]=1$. Pentru $i=2,...,NrDivizori$, iniţializăm $nmin[i]=+\infty$. Considerăm apoi toţi divizorii $dv[j]$ ($1 \leq j \leq i-1$) şi dacă $dv[j]$ este divisor al lui $dv[i]$, atunci calculăm numărul de suţinători necesari dacă cei $dv[i]$ alegători ar fi împărţiţi în $dv[j]$ grupe: acest număr este $NS(i,j)=((dv[i] \text{ div } dv[j]) \text{ div } 2)+1) \cdot nmin[j]$; setăm apoi $nmin[i]=min\{nmin[i], NS(i,j)\}$.

$nmin[NrDivizori]$ reprezintă răspunsul problemei.

Problema 1-12. Drumuri palindromice (USACO 2002, enunţ modificat)

Se dă un graf neorientat cu N ($1 \le N \le 1000$) noduri. Fiecare nod are asociată o literă dintr-un alfabet cu C simboluri ($l(i)$ este litera asociată nodului i). Determinaţi câte drumuri palindromice de lungime fix L ($1 \le L \le 500$ există. Un drum este palindromic dacă, parcurgându-l dintre-un sens şi concatenând simbolurile asociate nodurilor de pe drum, obţinem un palindrom. Un drum poate conţine acelaşi nod de mai multe ori.

Soluţie: O primă soluţie este următoarea. Vom calcula matricea $NPAL(i,j,k)$=numărul de drumuri palindromice de lungime k dintre nodurile i şi j. Avem $NPAL(i,i,1)=1$ şi $NPAL(i,j,2)=1$, dacă $l(i)=l(j)$ şi nodurile i şi j sunt adiacente în graf.

Pentru fiecare lungime $3 \le k \le L$ vom calcula $NPAL(i,j,k)$ după cum urmează. Dacă $l(i) \ne l(j)$, atunci $NPAL(i,j,k)=0$. Altfel, iniţializăm $NPAL(i,j,k)=0$, apoi considerăm toţi vecinii i' ai nodului i şi toţi vecinii j' ai nodului j, astfel încât $l(i')=l(j')$ şi setăm $NPAL(i,j,k)=NPAL(i,j,k)+NPAL(i',j',k-2)$. Soluţia este suma valorilor $NPAL(*,*,L)$. Acest algoritm are complexitatea $O(N^2 \cdot L \cdot D^2)$, unde D este numărul maxim de vecini ai unui nod din graf. Când D este mare, putem îmbunătăţi algoritmul după cum urmează.

Calculăm, în plus, $NPALaux(i,j,k)$=numărul de drumuri de lungime k, cu proprietatea că încep în nodul i, se termină în nodul j, iar subdrumul format din bucata de drum ce începe imediat după nodul i şi se termină la nodul j este palindromic. Pentru $3 \le k \le L$, definiţiile se schimbă după cum urmează. Pentru a calcula $NPALaux(i,j,k)$ iniţializăm valoarea cu 0 şi apoi considerăm toţi vecinii i' ai nodului i, setând $NPALaux(i,j,k)=NPALaux(i,j,k)+NPAL(i',j,k-1)$.

$NPAL(i,j,k)$ (cu $l(i)=l(j)$) va fi acum egal cu suma valorilor $NPALaux(i,j',k-1)$, cu j' un vecin al nodului j. Complexitatea s-a redus la $O(N^2 \cdot L \cdot D)$.

Problema 1-13. Schimb valutar (Olimpiada Naţională de Informatică, Croaţia, 2001)

Gigel are X euro. El a aflat dinainte, pentru fiecare din următoarele N ($1 \le N \le 100.000$) zile, rata de schimb dintre euro şi dolar. Mai exact, pentru fiecare zi i ($1 \le i \le N$) se ştie că 1 $dolar=DolEuro(i)$ euro (dacă se face conversia din dolari în euro) şi 1 $euro=EuroDol(i)$ dolari (dacă se face conversia din euro în dolari). Determinaţi cea mai mare sumă (în euro) pe care o poate avea Gigel după cele N zile. În fiecare zi, el poate schimba dolarii în euro sau euro în dolari (nu neapărat pe toţi), sau poate să nu efectueze nicio tranzacţie de schimb valutar.

Soluţie: Vom calcula $Dmax(i)$ şi $Emax(i)$, suma maximă de dolari (respectiv, euro), pe care o poate avea Gigel la sfârşitul primelor i zile. Avem $Dmax(0)=0$ şi $Emax(0)=X$. Pentru $1 \le i \le N$ avem: $Dmax(i)=max\{Dmax(i-1), Emax(i-1) \cdot EurDol(i)\}$ şi $Emax(i)=max\{Emax(i-1), Dmax(i-1) \cdot DolEur(i)\}$. $Emax(N)$ este suma maximă în euro pe care o poate avea Gigel la sfârşitul celor N zile.

Problema 1-14. Tăiere cu maximizarea sumei

Se dă un şir de N ($1 \le N \le 100.000$) valori întregi: $a(1), \ldots, a(N)$ ($-1000 \le a(i) \le 1000$; $1 \le i \le N$). Eliminaţi K ($1 \le K \le 30$) secvenţe (disjuncte) de câte x ($1 \le K \cdot x \le N$) numere (consecutive), astfel încât suma numerelor rămase în secvenţă să fie maximă.

Soluţie: Maximizarea sumei elementelor rămase este echivalentă cu minimizarea sumei elementelor eliminate. Vom calcula *Smin(i,j)*=suma minimă a elementelor eliminate, dacă dintre primele *i* elemente am eliminat *j* secvenţe.

Avem $Smin(i,0)=0$ şi $Smin(i,j\cdot x>i)=+\infty$ ($0\leq i\leq N$). Pentru $i\geq 1$ şi $1\leq j\leq min\{i\ div\ x, K\}$, avem: $Smin(i,j)=min\{Smin(i-1,j),Smin(i-x,j-1)+Sum(i-x+1,i)\}$.

Sum(u,v) reprezintă suma elementelor dintre poziţiile *u* şi *v* (inclusiv). $Sum(u,v)=SP(b)-SP(a-1)$, unde *SP(q)*=suma primelor *q* elemente ($SP(0)=0$ şi $SP(1\leq q\leq N)=SP(q-1)+a(q)$). Complexitatea algoritmului este $O(N\cdot K)$.

Problema 1-15. Mesaj

Se dă un şir *S* format din *N* caractere ($1\leq N\leq 1.000$) şi o listă de *M* cuvinte ($0\leq M\leq 1.000$) de lungime cel mult *L* ($1\leq L\leq 20$). Determinaţi numărul minim de caractere ce trebuie eliminate din şirul *S*, astfel încât şirul rămas să poată fi reprezentat ca o concatenare a unor cuvinte din lista de cuvinte (în cadrul concatenării, un cuvânt poate apărea de *0*, *1*, sau mai multe ori). Cuvintele sunt formate din caractere ale unui alfabet *A* ce conţine cel mult *Q* litere ($1\leq Q\leq 1.000$).

Soluţie: Vom nota prin *S(i)* caracterul de pe poziţia *i* din şirul *S*, şi prin *S(i..j)* subşirul dintre poziţiile *i* şi *j* ale şirului *S*. Pentru început, vom parcurge şirul *S* de la sfârşit către început (de la dreapta la stânga) şi vom calcula, pentru fiecare poziţie *i* şi fiecare caracter *c*, valoarea *Next(i,c)*=cea mai din stânga poziţie $j\geq i$, astfel încât $S(j)=c$.

Pentru $i=N$ vom avea $Next(N, c\neq S(N))=N+1$ şi $Next(N, S(N))=N$. Pentru $1\leq i\leq N-1$, vom avea: $Next(i, S(i))=i$ şi $Next(i, c\neq S(i))=Next(i+1, c)$. Aşadar, aceste valori se pot calcula în timp $O(N\cdot Q)$ (cu *i* variind descrescător, de la *N* către *1*).

Vom calcula apoi *Nmin(i)*=numărul minim de caractere ce trebuie eliminate din *S(i..N)*, pentru ca şirul rămas (*S(i..N)* fără caracterele eliminate) să poată fi scris ca o concatenare de cuvinte din lista dată. Avem $Nmin(N+1)=0$. Pentru $1\leq i\leq N$ (în ordine descrescătoare a lui *i*), avem următoarele situaţii:

(1) nu începe niciun cuvânt la poziţia *i*; în acest caz, avem $Nmin_{cand}(i, 0)=1+Nmin(i+1)$;

(2) la poziţia *i* începe cuvântul *j* ($1\leq j\leq M$).

Pentru cazul *2*, va trebui să determinăm dacă cuvântul *j* chiar se regăseşte ca un subşir de caractere (nu neapărat consecutive) în *S(i..N)*. Pentru aceasta, vom parcurge cu un contor *k* toate poziţiile din cuvântul *j* (*Cuv(j)*); $1\leq k\leq |Cuv(j)|$ (*|Cuv(j)|*=numărul de caractere ale cuvântului *Cuv(j)*). În cadrul parcurgerii vom reţine câte o poziţie *poz*, reprezentând cea mai din stânga poziţie pe care ar putea apărea caracterul *Cuv(j,k)* (al *k*-lea caracter din *Cuv(j)*).

Iniţial avem $poz=i$. Pentru $k=1,...,|Cuv(j)|$ efectuăm următoarele acţiuni: $poz=Next(poz, Cuv(j,k))+1$; dacă $poz\geq N+2$, atunci întrerupem ciclul al cărui contor este *k*.

Dacă la sfârşitul parcurgerii caracterelor din *Cuv(j)* avem $poz\leq N+1$, atunci cuvântul *Cuv(j)* se termină, cel mai devreme, pe poziţia *poz-1*. Aşadar, vom avea $Nmin_{cand}(i,j)=poz-i-|Cuv(j)|+Nmin(poz)$. Altfel, dacă $poz>N+1$, setăm $Nmin_{cand}(i,j)=+\infty$. *Nmin(i)* va fi egal cu $min\{Nmin_{cand}(i,j)|0\leq j\leq M\}$.

Problema poate fi extinsă la cazul în care fiecare caracter *c* ($1\leq c\leq Q$) are un cost de eliminare $Cost(c)\geq 0$. În acest caz, vom calcula uşor diferit valorile candidat $Nmin_{cand}(i,j)$ ($1\leq i\leq N$; $0\leq j\leq M$). $Nmin_{cand}(i,0)=Cost(S(i))+Nmin(i+1)$. Vom calcula în prealabil (la început) sumele prefix *SP(u)*: $SP(0)=0$ şi $SP(1\leq u\leq N)=Cost(S(u))+SP(u-1)$. De asemenea, pentru fiecare cuvânt *Cuv(j)*, vom calcula la început *Scost(j)*=suma costurilor *Cost(Cuv(j,k))* ($1\leq k\leq |Cuv(j)|$). Cu aceste valori calculate, şi calculând poziţia *poz* folosind acelaşi algoritm

prezentat anterior, vom avea $Nmin_{cand}(i,j)=SP(poz-1)-SP(i-1)-Scost(j)+Nmin(poz)$ (pentru cazul $1\leq j\leq M$ şi $poz\leq N+1$).

Problema 1-16. Tetris

Se dă o tablă de tetris alcătuită din N ($1\leq N\leq 3$) coloane şi înălţime infinită. De asemenea, se dă o secvenţă de M piese de tetris ce trebuie aşezate, în ordine, pe tablă. O piesă de tetris este o componentă conexă de pătrăţele, ce poate fi rotită şi translatată. Dorim să amplasăm toate piesele pe tablă astfel încât înălţimea maximă ocupată pe oricare dintre coloane să fie minimă.

Soluţie: O primă soluţie este următoarea. Vom calcula matricea $OK(h(1), ..., h(N), j)=1$, dacă coloana i este ocupată până la înălţimea $h(i)$ (eventual cu găuri) ($1\leq i\leq N$) şi am considerat deja primele j ($0\leq j\leq M$) piese.

Începem cu $OK(0, ..., 0, 0)=1$ şi $OK(h(1), ..., h(N), 0)=0$ dacă cel puţin o valoare $h(i)$ este mai mare decât 0. Considerăm apoi, pe rând, fiecare stare $(h(1), ..., h(N), j)$ în ordine crescătoare a lui j (şi în orice ordine pentru j egal) pentru care $OK(h(1), ..., h(N), j)=1$ ($1\leq j\leq M-1$).

Vom considera fiecare posibilitate de a translata şi roti piesa $j+1$ şi apoi o vom "amplasa" pe tablă. După selectarea translaţiei (practic, se alege cea mai din stânga coloană pe care o va ocupa piesa) şi a rotaţiei, pentru fiecare coloană j ocupată de o pătrăţică a piesei, vom calcula $hjos(j)$ ($hsus(j)$) = diferenţa în modul între linia cea mai de jos (sus) a unui pătrăţel al piesei de pe coloana j şi linia celui mai de sus pătrăţel al piesei (indiferent de coloana pe care se află acesta). În urma amplasării, noile înălţimi ale coloanelor vor fi $h'(1), ..., h'(N)$. Dacă o coloană j nu este ocupată de niciun pătrăţel al piesei, atunci $h'(j)=h(j)$. Pentru coloanele j ocupate de piesă vom calcula $hmax=max\{hjos(j)+h(j) \mid j$ este o coloană ocupată de cel puţin un pătrăţel al piesei$\}$; pentru o astfel de coloană j vom avea $h'(j)=hmax-hsus(j)$.

Vom seta $OK(h'(1), ..., h'(N), j+1)=1$. La final, răspunsul va fi $min\{max\{h(1), ..., h(N)\} \mid OK(h(1), ..., h(N), N)=1\}$. Considerând că fiecare piesă are dimensiune $O(1)$, această soluţie are un ordin de complexitate de $O(M^N \cdot N \cdot M)$.

O îmbunătăţire este următoarea. Vom calcula $hmin(h(1), ..., h(N-1), j)$=înălţimea minimă pentru $h(N)$, dacă înălţimile coloanelor $1, ..., N-1$ sunt $h(1), ..., h(N-1)$ şi am considerat primele j piese. Vom avea $hmin(0, ..., 0, 0)=0$ şi $hmin(h(1), ..., h(N-1), 0)=+\infty$ dacă avem cel puţin o înălţime $h(i)>0$.

La fel ca înainte, vom considera stările $(h(1), ..., h(N-1), j)$ în aceeaşi ordine ($1\leq j\leq M-1$). Vom considera fiecare translaţie şi rotaţie a piesei $j+1$. Considerând $h(N)=hmin(h(1), ..., h(N-1), j)$, putem folosi acelaşi procedeu ca şi în soluţia anterioară pentru a obţine valorile $h'(1), ..., h'(N)$. Apoi vom seta $hmin(h'(1), ..., h'(N-1), j+1)=min\{h'(N), hmin(h'(1), ..., h'(N-1), j+1)\}$ (iniţial, toate valorile $hmin(*, ..., *)$ se consideră a fi $+\infty$, cu excepţia lui $hmin(0, ..., 0)$).

Răspunsul va fi $min\{hmin(h(1), ..., h(N-1), N)\}$. Complexitatea acestei soluţii este $O(M^{N-1} \cdot N \cdot M)$ (exponentul a fost redus cu 1 unitate).

Problema 1-17. Vectori de Teleportare pentru K Persoane

K ($1\leq K\leq 5$) persoane se află într-un cub d-dimensional ($1\leq d\leq 5$) ce constă din N ($1\leq N\leq 100.000^{1/(d\cdot(K-1)+1)}$) pătrăţele în fiecare dimensiune (numerotate de la 1 la N). Persoana i ($1\leq i\leq K$) se află iniţial la poziţia $(xs(i,1), ..., xs(i,d))$ şi trebuie să ajungă în poziţia $(xf(i,1), ..., xf(i,d))$. Se dă un şir de M ($1\leq M\leq 100$) vectori d-dimensionali de *teleportare*; al j-lea vector

este $(v(j,1), ..., v(j,d))$, cu $-N+1 \le v(j,q) \le N-1$ ($1 \le q \le d$). Pentru fiecare vector din şir, în ordinea în care sunt daţi, se ştie că exact una din cele K persoane va efectua o *teleportare* corespunzătoare acelui vector. Dacă o persoană se află în poziţia $(x(1), ..., x(d))$ şi se teleportează folosind vectorul $(v(j,1), ..., v(j,d))$, atunci noua sa poziţie va fi $(x(1)+v(j,1), ..., x(d)+v(j,d))$, cu condiţia ca această poziţie să se afle în interiorul cubului.

În interiorul cubului anumite poziţii sunt blocate. O persoană nu se poate teleporta într-o poziţie blocată. Poziţiile iniţiale ale persoanelor nu sunt blocate.

Determinaţi dacă este posibil ca fiecare persoană să ajungă în poziţia sa finală, folosind şirul de M vectori de teleportare dat.

Soluţie: O primă soluţie constă în a calcula următoarea matrice: $OK(j, xc(1,1), ..., xc(1,d), ..., xc(i,1), ..., xc(i,d), ..., xc(K,1), ..., xc(K,d))=1$, dacă după folosirea primilor j vectori, fiecare persoană i poate ajunge în poziţia $(xc(i,1), ..., xc(i,d))$ ($1 \le i \le K$) (şi 0, altfel).

Iniţial avem $OK(0, xs(1,1), ..., xs(1,d), ..., xs(K,1), ..., xs(K,d))=1$. Pentru a calcula $OK(j, S=(xc(1,1), ..., xc(1,d), ..., xc(K,1), ..., xc(K,d)))$ vom considera, pe rând, fiecare persoană i şi vom încerca dacă aceasta poate fi cea care se teleportează folosind vectorul j. Calculăm poziţia anterioară $xc'(i,q)=xc(i,q)-v(j,q)$ ($1 \le q \le d$). Dacă $OK(j-1, S'(i)=(xc(p, 1), ..., xc(p,d)$ $(1 \le p \le i-1), xc'(i,1), \dots, xc'(i,d), xc(p',1), ..., xc(p', d)$ $(i+1 \le p' \le K)))=1$, atunci vom seta $OK(j, S)=1$.

Dacă pentru nicio persoană i ($1 \le i \le K$) nu găsim $OK(j-1, S'(i))=1$, atunci $OK(j,S)=0$. Vom considera, prin definiţie, că $OK(j, xc(p,q)$ $(1 \le p \le K, 1 \le q \le d))=0$ dacă vreuna din coordonatele $xc(p',q')$ se află în afara cubului sau dacă vreo poziţie $(xc(i,1), ..., xc(i,d))$ este blocată.

La final vom verifica dacă $OK(M, xf(p,q)$ $(1 \le p \le K, 1 \le q \le d))=1$ şi, dacă da, vom reconstitui soluţia folosind valorile deja calculate. Acest algoritm are complexitatea $O(M \cdot N^{d \cdot K+1})$ şi calculează $O(M \cdot N^{d \cdot K})$ stări. Se observă uşor că se doreşte ca complexitatea recomandată să fie $O(M \cdot d+M \cdot N^{d \cdot (K-1)+1})$. Aşadar, soluţia găsită este cu un factor de $O(N^d)$ mai slabă.

Pentru a optimiza complexitatea algoritmului vom calcula sumele parţiale $SV(j,q)$ ($1 \le p \le M$, $1 \le q \le d$). Avem $SV(0,*)=0$ şi $SV(1 \le j \le M, q)=SV(j-1, q)+v(j,q)$. Observăm acum că, dacă după folosirea a j vectori de teleportare, persoanele i ($1 \le i \le K-1$) se află în poziţiile $xc(i,q)$ ($1 \le q \le d$), atunci putem determina în mod unic poziţia persoanei K. Astfel, vom calcula $Scur(q)=(xc(1,q)+...+xc(K-1,q))-(xs(1,q)+...+xs(K-1,q))$ ($1 \le q \le d$) (sau 0, dacă $K=1$). Poziţia în care se află persoana K este $xc(K,q)=xs(K,q)+SV(j,q)-Scur(q)$ $(1 \le q \le d)$. Algoritmul nu se modifică cu nimic, cu excepţia faptului că nu vom mai menţine şi poziţia persoanei K în cadrul stării din matricea OK, ea fiind dedusă, de fiecare dată, din poziţiile persoanelor $1, ..., K-1$. În acest fel, complexitatea algoritmului s-a redus la $O(M \cdot d+M \cdot N^{d \cdot (K-1)+1})$.

Problema 1-18. Cai multicoloraţi (TIMUS, enunţ modificat)

Se dau N ($1 \le N \le 500$) cai aşezaţi în ordine, unul după altul (de la calul numerotat cu 1, la calul numerotat cu N). Fiecare cal i ($1 \le i \le N$) are o culoare $c(i)$ ($1 \le c(i) \le K$; $2 \le K \le 20$) şi o forţă $f(i)$ ($0 \le f(i) \le 100$).

Caii trebuie împărţiţi în Q ($1 \le Q \le min\{N, 100\}$) intervale (de cai aşezaţi pe poziţii consecutive), în aşa fel încât fiecare cal face parte dintr-un interval şi fiecare interval conţine cel mult $CMAX$ cai. Toţi caii dintr-un interval vor fi duşi în acelaşi grajd. Să presupunem că „suma” forţelor cailor de culoarea j dintr-un interval $[a,b]$ este $sf(j)$ ($1 \le j \le K$) (obţinută prin aplicarea funcţiei comutative $csum(j) \in \{+,max\}$ asupra valorilor forţelor cailor de culoarea j din interval). Agresivitatea cailor din intervalul respectiv este egală cu *produsul* valorilor $sf(j)$ $(sf(1) \cdot sf(2) \cdot ... \cdot sf(K))$.

Determinaţi o împărţire a cailor în intervale astfel încât agresivitatea totală (obţinută prin aplicarea funcţiei comutative $fsum \in \{+, max\}$ asupra agresivităţilor din fiecare interval) să fie minimă.

Exemplu:

N=6; K=2; Q=3 **fsum=+; csum(*)=+** **c(1)=1; f(1)=1** **c(2)=1; f(2)=1** **c(3)=2; f(3)=1** **c(4)=1; f(4)=1** **c(5)=2; f(5)=1** **c(6)=1; f(6)=1**	**Agresivitatea totală minimă este 2. Cele Q=3 intervale formate pot fi: [1,2], [3,5], [6,6]. Agresivitatea în fiecare din cele 3 intervale este: 0, 2, 0.**

Soluţie: Vom calcula $Amin(i,j)$=agresivitatea totală minimă pentru a împărţi primii i cai în j intervale. Avem $Amin(0,0)=0$ şi $Amin(i,j>i)=+\infty$. Pentru a calcula $Amin(i,j)$ va trebui să alegem al j-lea interval. Despre acest interval ştim că se termină la calul i, dar nu ştim la ce cal începe. Vom iniţializa un vector $sf(col)$ ($1 \leq col \leq K$) cu valoarea 0. Vom menţine, de asemenea, $Pcol$=produsul elementelor din vectorul sf.

Vom considera apoi fiecare poziţie p de început a celui de-al j-lea interval, în sens descrescător, începând de la $p=i$ şi terminând la $p=max\{j, i-CMAX+1\}$. Vom iniţializa $Amin(i,j)=+\infty$. Când ajungem la o valoare nouă a lui p, vom seta $sf(c(p))=csum(c(p))(sf(c(p)), f(p))$. Vom calcula apoi noua valoare a lui $Pcol$. Putem realiza acest lucru în timp $O(K)$ (parcurgând întreg vectorul sf) sau, mai eficient, în timp $O(1)$.

Vom menţine, în plus, numărul $nzero$ de element egale cu zero din vectorul sf (iniţial, $nzero=K$) şi Pnz, produsul elementelor nenule din vectorul sf (iniţial, $Pnz=1$). Înainte să modificăm $sf(c(p))$ cu $f(p)$, dacă $sf(c(p))=0$ şi $f(p)>0$, atunci decrementăm $nzero$ cu 1. Dacă $nzero$ tocmai a scăzut cu 1 unitate, atunci setăm $sf(c(p))=csum(c(p))(sf(c(p)), f(p))$ (realizăm modificarea) şi setăm $Pnz=Pnz \cdot sf(c(p))$. Dacă $nzero$ nu a scăzut şi $sf(c(p))>0$, atunci fie $sfold(c(p))$ valoarea lui $sf(c(p))$ dinaintea modificării. Setăm $Pnz=Pnz/sfold(c(p)) \cdot sf(c(p))$. Dacă $nzero>0$, atunci $Pcol=0$; altfel, $Pcol=Pnz$.

După ce avem calculată valoarea lui $Pcol$ corespunzătoarea poziţiei p, setăm $Amin(i,j)=min\{Amin(i,j), fsum(Amin(p-1,j-1), Pcol)\}$.

Valoarea $Amin(N,Q)$ este agresivitatea totală minimă. Împărţirea în intervale poate fi şi ea determinată uşor, dacă pentru fiecare pereche (i,j) reţinem acea valoare a lui p pentru care s-a obţinut $Amin(i,j)$. Complexitatea algoritmului este $O(N^2 \cdot Q)$.

Observăm că putem generaliza problema. Putem folosi alte funcţii de agregare pentru:

(1) „suma" forţelor cailor de aceeaşi culoare dintr-un interval => putem folosi, de exemplu, suma, maximul sau minimul;

(2) costul unui interval => în loc de produsul forţelor cailor de culori diferite putem folosi suma, maximul, minimul, etc.

Multe dintre aceste combinaţii pot fi suportate modificând foarte puţin algoritmul prezentat (şi păstrând complexitatea $O(N^2 \cdot Q)$). De exemplu, dacă funcţia f ce determină costul unui interval este inversabilă (are o funcţie inversă), atunci, când trecem la o valoare p nouă, îi putem calcula valoarea în $O(1)$ (*valoare nouă=(valoare veche)* f^{-1} $sfold(c(p))$ f $sf(c(p))$). De asemenea, dacă funcţia pentru „suma" forţelor cailor de aceeaşi culoare dintr-un interval ($sf(c(p))$) este *max* sau *min* şi funcţia f pentru costul unui interval (pe baza costurilor asociate fiecărei culori) este aceeaşi (*max* sau *min*), avem: *valoare nouă=f(valoare veche,*

$sf(c(p)))$. Tot pentru cazul *max/min*, putem menţine un *max/min*-heap cu valorile $sf(c(p))$ pe măsură ce decrementăm valoarea lui p (când se modifică $sf(c(p))$ ştergem din heap valoarea veche şi o introducem pe cea nouă). În aceste cazuri, valoarea nouă a costului unui interval se poate obţine extragând elementul din vârful heap-ului, în timp $O(log(K))$ (ajungând la o complexitate de $O(N^2 \cdot Q \cdot log(K))$). Pentru alte combinaţii, însă, va trebui să recalculăm costul corespunzător unui interval în timp $O(K)$ (pentru fiecare valoare a lui p), ajungând la o complexitate de $O(N^2 \cdot Q \cdot K)$.

Problema 1-19. Vinuri (Olimpiada Naţională de Informatică, România 1997)

La un depozit specializat din Recaş sosesc, pe rând, N ($1 \le N \le 300$) clienţi care solicită fiecare o cantitate dată de Cabernet (clientul i cere $L(i)$ litri, $1 \le i \le N$, $1 \le L(i) \le 100$), oferind o sumă pentru achiziţionarea întregii cantităţi cerute ($S(i)$, $0 \le S(i) \le 1.000.000$). În butoiul din magazinul de deservire (considerat de capacitate nelimitată) nu se găseşte iniţial nicio picătură de vin. Dan Şeptică, administratorul depozitului, are un algoritm propriu de deservire a clienţilor: în funcţie de ceea ce are în butoi, dar şi de inspiraţia de moment, el poate să răspundă:

- Vă ofer întreaga cantitate cu cea mai mare placere !

sau

- Nu vă pot oferi acum ceea ce doriţi, mai reveniţi cu solicitarea altă dată!

Pe clientul servit îl eliberează de grija banilor corespunzători costului licorii cumpărate, iar pe cel refuzat îl salută politicos şi are grijă ca, imediat ce a plecat clientul (i), să coboare şi să aducă în butoiul din magazin exact cantitatea solicitată de clientul respectiv (cel ce nu a fost servit), adică $L(i)$.

Cunoscând cele N cantităţi cerute de clienţi, să se determine un mod de a răspunde solicitărilor astfel încât, în final, suma încasată să fie maximă.

Soluţie: Observăm că, dacă toţi clienţii ar fi refuzaţi, cantitatea maximă de vin ce s-ar putea afla în butoi este cel mult egală cu $LMAX=30.000$. Aşadar, vom putea calcula valorile $Smax(i,j)$=suma maximă ce poate fi încasată, dacă după ce au sosit primii i clienţi au rămas j litri în butoi.

Iniţial, $Smax(0,0)=0$ şi $Smax(0, j>0)=-\infty$. Pentru $i \ge 1$ avem: pentru $0 \le j \le L(i)-1$, $Smax(i,j)=S(i)+Smax(i-1, j+L(i))$; pentru $L(i) \le j \le LMAX$, $Smax(i,j)=max\{S(i)+Smax(i-1, j+L(i)), Smax(i-1, j-L(i))\}$.

Rezultatul este $max\{Smax(N,j) | 0 \le j \le LMAX\}$. Complexitatea algoritmului este $O(N \cdot LMAX)$.

Problema 1-20. Funcţii de optimizare într-o secvenţă

Se dă o secvenţă de N ($1 \le N \le 100.000$) numere $v(i)$ ($1 \le i \le N$). Fiecare poziţie i are asociat un interval de poziţii $[l(i), r(i)]$ ($1 \le l(i) \le r(i) \le i-1$), o valoare $z(i)$ şi o funcţie $f(i)(x)$ crescătoare. Dorim să determinăm o secvenţă (*(a)* de K numere (unde K poate aparţine unei mulţimi $SK \subseteq \{1,...,N\}$) ; *(b)* fără limită de numere) $u(1)<u(2)<...<u(K)$, astfel încât: vom defini $q(u(1))=z(u(1))$ şi $q(u(j))=f(u(j))(q(u(j-1)))$ ($2 \le j \le K$); în plus, $u(j)$ se află în intervalul $[l(u(j+1)), r(u(j+1))]$. Dorim ca valoarea $q(u(K))$ să fie minimă.

Soluţie: Pentru cazul *a)* vom calcula valorile $qmin(i,j)$=cea mai mică valoare, dacă în submulţimea pe care o selectăm, avem $u(j)=i$. $qmin(i,1)=z(i)$ ($1 \le i \le N$).

Pentru $2 \le j \le K$, vom considera valorile i în ordine crescătoare. $qmin(i,j)=f(i)(qm(i,j)=min\{+\infty, min\{qmin(p,j-1) | l(i) \le p \le r(i)\}\})$.

Pentru calcularea valorii $qm(i,j)$ putem considera toate poziţiile p din intervalul $[l(i), r(i)]$, pentru a obţine o complexitate $O(N^2 \cdot K)$. Răspunsul este $min\{q(i,j)|j \in SK\}$. Dacă înainte de a calcula valorile $qmin(*,j)$ construim un arbore de intervale peste valorile $qmin(*,j-1)$, atunci putem reduce complexitatea la $O(N \cdot K \cdot log(N))$. Pentru a calcula $qm(i,j)$ vom folosi o interogare de tipul „range minimum query" pe intervalul $[l(i), r(i)]$, la care poate fi obţinut răspunsul în timp $O(log(N))$. Dacă folosim tehnica RMQ [Bender-RMQLCA2000] peste valorile $qmin(*,j-1)$ (preprocesare $O(N \cdot log(N))$ şi găsirea minimului dintr-un interval $[a,b]$ în timp $O(1)$), obţinem aceeaşi complexitate, dar fără utilizarea unui arbore de intervale. Acestă soluţie poate ajunge şi la $O(N \cdot K)$, deoarece etapa de preprocesare a RMQ poate fi redusă (folosind nişte metode mai complicate), la $O(N)$.

Dacă avem $l(i) \leq l(i+1)$ şi $r(i) \leq r(i+1)$ ($1 \leq i \leq N-1$), atunci putem folosi un deque (double-ended queue). Deque-ul va reţine perechi *(valoare,poziţie)*. Când vrem să adaugăm o pereche *(val,poz)* la sfârşitul deque-ului, vom efectua următorii paşi: cât timp deque-ul nu este gol şi ultima pereche *(val',poz')* din deque are proprietatea $val' \geq val$, vom şterge perechea *(val', poz')* din deque. Când trecem la o nouă valoare a lui j, golim deque-ul. Când ajungem la o poziţie i, efectuăm următorii paşi:

(1) cât timp deque-ul nu e gol şi perechea *(val,poz)* de la începutul deque-ului are proprietatea $poz < l(i)$, ştergem această pereche de la începutul deque-ului;

(2) adăugăm, pe rând, la sfârşitul deque-ului, perechile ($val=qmin(p,j-1)$, $poz=p$), cu $r(i-1)+1 \leq p \leq r(i)$ (înaintea fiecărei adăugări efectuăm paşii descrişi anterior) ; *(3)* dacă deque-ul e gol, atunci $qmin(i,j)=+\infty$; altfel, fie *(val,poz)* perechea de la începutul deque-ului; vom seta $qmin(i,j)=f(i)(val)$. Utilizarea deque-ului asigură o complexitate $O(N \cdot K)$.

Pentru cazul *(b)* vom renunţa la indicele j din cadrul dinamicii: vom calcula valorile $qmin(i)$=cea mai mică valoare, dacă i este ultima poziţie din cadrul submulţimii selectate. Avem $qmin(1)=z(1)$. Pentru $2 \leq i \leq N$, vom avea $qmin(i)=min\{z(i), qm(i)=min\{qmin(p)|l(i) \leq p \leq r(i)\}\}$. O implementare directă (care consideră, pe rând, fiecare poziţie p), are o complexitate $O(N^2)$.

Putem folosi şi de această dată un arbore de intervale. Interogările sunt aceleaşi. Diferenţa este dată de faptul că atunci când calculăm o valoare $qmin(i)$, atribuim această valoare frunzei i din arbore (iniţial, fiecare frunză va avea asociată valoare $+\infty$) – acest tip de modificare se numeşte „point update" [Andreica-ISPDC2008]. Fiecare interogare şi actualizare pot fi realizate în timp $O(log(N))$.

Dacă avem proprietăţile $l(i) \leq l(i+1)$ şi $r(i) \leq r(i+1)$, atunci putem folosi din nou un deque. Diferenţele faţă de algoritmul anterior sunt următoarele. Iniţial, deque-ul este gol. Când ajungem la o poziţie i, ştergem în mod repetat perechea *(val,poz)* de la începutul deque-ului cât timp $val < l(i)$ şi deque-ul nu e gol, apoi inserăm perechile $(val=qmin(p), poz=p)$, cu $r(i-1)+1 \leq p \leq r(i)$, la sfârşitul deque-ului (înaintea fiecărei inserări efectuăm paşii descrişi anterior, pentru a ne asigura că valorile din deque sunt sortate în ordine crescătoare). Dacă deque-ul e gol, atunci $qmin(i)=z(i)$; altfel, fie *(val,poz)* prima pereche din deque: $qmin(i)=min\{z(i), f(i)(val)\}$.

Răspunsul este $min\{qmin(i)|1 \leq i \leq N\}$. Complexitatea algoritmului este $O(N \cdot log(N))$ (dacă folosim arbori de intervale) şi, respectiv, $O(N)$, dacă folosim (şi suntem în situaţia de a putea folosi) deque-ul.

Problema 1-21. Submulţime de noduri cu restricţii

Se dă un graf neorientat cu N ($1 \leq N \leq 1.000$) noduri. Fiecare nod i al grafului are o pondere $w(i)$ ($1 \leq i \leq N$). Notăm prin $NV(i)$ mulţimea nodurilor adiacente cu nodul i. Fiecare nod i are

asociate *2* seturi de submulţimi: $S(i,1)$ şi $S(i,2)$, având $ns(i,1)$, respectiv $ns(i,2)$ submulţimi *valide* fiecare (submulţimile sunt identificate prin $S(i,p,j)$ ($1 \le j \le ns(i,p)$; $p=1,2$), $S(i,p,j) \subseteq NV(i)$, iar fiecare element al lui $S(i,p,j)$ este strict mai mic decât i).

Seturile de submulţimi $S(i,p)$ pot fi date şi sub formă implicită (de ex., $S(i,p)$ conţine toate submulţimile de vecini ai nodului i având un număr x ($0 \le x \le |NV(i)|$) de elemente ce aparţin unei mulţimi de numere $XV(i,p)$). Determinaţi o submulţime SN de noduri ale grafului având pondere totală maximă (minimă), astfel încât fiecare nod i ($1 \le i \le N$) să respecte următoarele proprietăţi:

- dacă nodul i face parte din SN, atunci trebuie să existe cel puţin o submulţime $S(i,1,j)$, astfel încât toate nodurile din $S(i,1,j)$ fac parte şi ele din SN
- dacă nodul i nu face parte din SN, atunci trebuie să existe cel puţin o submulţime $S(i,2,j)$, astfel încât toate nodurile din $S(i,2,j)$ fac parte din SN

Se ştie că pentru orice muchie (a,b) din graf are loc proprietatea: $|a-b| \le K$ ($1 \le K \le 11$).

Soluţie: Vom parcurge nodurile în ordine, de la 1 la N, şi vom calcula un tabel $Wopt(i,ST)$=ponderea maximă (minimă) a unei submulţimi a nodurilor $\{1,2,...,i\}$ astfel încât: *(1)* toate nodurile $1 \le j \le i$ respectă proprietăţile; şi *(2)* ST este o submulţime a nodurilor $\{i-K+1, ..., i\}$ care indică care dintre aceste noduri fac parte din SN. Răspunsul optim este $opt\{Wopt(N, *)\}$ (unde $opt=max$ sau $opt=min$, în funcţie de caz – *max* dacă urmărim maximizarea ponderii şi *min* altfel).

Iniţial avem $Wopt(0,\{\})=0$. Pentru $1 \le i \le N$ vom folosi următoarele formule:

(1) $Wopt(i, (S' \cup \{i\}) \setminus \{i-K\})=opt\{Wopt(i-1, S') \mid \exists\ 1 \le p \le |S(i,1)|\ a.î.\ S(i,1,p) \subseteq S'\}$

(2) $Wopt(i, S' \setminus \{i-K\})=opt\{Wopt(i-1, S') \mid \exists\ 1 \le p \le |S(i,2)|\ a.î.\ S(i,2,p) \subseteq S'\}$

Capitolul 2. Greedy şi Euristici

Problema 2-1. Colorarea intervalelor (TIMUS)

Se dau N ($1 \leq N \leq 100.000$) intervale $[A(i),B(i)]$ ($1 \leq i \leq N$). Se ştie că reuniunea acestor intervale este intervalul $[0,L]$. Dorim să colorăm o parte dintre cele N intervale (şi să le lăsăm pe celelalte necolorate), astfel încât suma lungimilor porţiunilor din intervalul $[0,L]$ care sunt incluse într-un singur interval colorat să fie cel puţin $2/3 \cdot L$.

Soluţie: Vom sorta intervalele după capătul stânga, astfel încât să avem $A(o(1)) \leq A(o(2)) \leq \ldots \leq A(o(N))$. Vom menţine coordonata dreapta *xr* a ultimului interval colorat. Iniţial, $xr=0$. Vom parcurge intervalele în ordinea sortată. Pe măsură ce parcurgem intervalele, vom menţine un indice *icand* (iniţial 0) şi suma maximă cu care poate creşte suma lungimilor porţiunilor acoperite de un singur interval (*smax*).

Când ajungem la un interval $o(i)$ ($1 \leq i \leq N$), întâi verificăm dacă $icand>0$ şi dacă $A(o(i))>xr$. Dacă da, atunci vom colora intervalul *icand*, vom seta $xr=B(icand)$ şi vom mări suma lungimilor porţiunilor acoperite de un singur interval colorat (*SL*) cu *smax*; apoi resetăm *icand* şi *smax* la 0.

După aceasta, vom considera intervalul $o(i)$ ca un posibil interval candidat pentru a fi colorat. Vom calcula $s(i)$=cu cât va creşte *SL* dacă intervalul $o(i)$ ar fi colorat. Dacă $A(o(i)) \geq xr$, atunci $s(i)=B(o(i))-A(o(i))$; altfel, dacă $B(o(i))>xr$, atunci $s(i)=(B(o(i))-xr)-(xr-A(o(i)))$; altfel, $s(i)=0$. Dacă $s(i)>smax$, atunci setăm $smax=s(i)$ şi $icand=o(i)$. La sfârşitul parcurgerii, dacă $icand>0$, vom colora intervalul *icand* şi vom aduna *smax* la *SL* (*SL* este 0 la începutul algoritmului).

Algoritmul descris este un algoritm euristic pentru maximizarea sumei porţiunilor incluse într-un singur interval colorat, care nu oferă niciun fel de garanţii. Totuşi, în practică, pentru multe cazuri, el se comportă destul de bine. Încercaţi să găsiţi un test pentru care algoritmul descris obţine un rezultat mai prost decât $2/3 \cdot L$.

Problema 2-2. Ordonarea înălţimilor (SGU)

Se dau N ($1 \leq N \leq 6.000$) înălţimi $h(1), \ldots, h(N)$. Înălţimile sunt numere reale între $1.950.000$ şi $2.050.000$. Media aritmetică a înălţimilor este $2.000.000$. Determinaţi (dacă există) o ordonare $o(1), \ldots, o(N)$ a înălţimilor ($h(o(1)), \ldots, h(o(N))$), astfel încât oricum am alege două poziţii diferite p şi q ($p<q$), suma înălţimilor dintre poziţiile p şi q inclusiv ($h(o(p))+h(o(p+1))+\ldots+h(o(q))$) să fie în intervalul $[2.000.000 \cdot (q-p+1)-100.000, 2.000.000 \cdot (q-p+1)+100.000]$.

Soluţie: Vom folosi următorul algoritm euristic. Pentru început, vom ordona înalţimile, astfel încât să avem $h(1) \leq h(2) \leq \ldots \leq h(N)$. Vom iniţializa două variabile $i=1$ şi $j=N$ şi vom menţine un sens *dir* cu valorile 0 sau 1 (iniţial, $dir=0$) şi suma S a înalţimilor adăugate până acum în cadrul ordonării (iniţial, $S=0$).

Apoi vom adăuga, pe rând, pe fiecare poziţie k de la 1 la N, câte o înălţime. La fiecare poziţie k, dacă $dir=0$, vom adăuga pe poziţia k înălţimea $h(i)$ (după care setăm $i=i+1$), iar dacă $dir=1$, adăugăm pe poziţia k înălţimea $h(j)$ (după care setăm $j=j-1$); apoi incrementăm S

cu valoarea înălţimii adăugate. Dacă $S<k\cdot 2.000.000$, atunci setăm $dir=1$; altfel, setăm $dir=0$. După aceasta, trecem la următoarea poziţie ($k+1$).

La final mai trebuie să verificăm dacă ordonarea obţinută respectă proprietăţile cerute. Să observăm că dacă există două poziţii p şi q pentru care suma înălţimilor dintre ele nu respectă proprietatea cerută, atunci una din perechile $(1,q)$ sau (p,N) nu va respecta, de asemenea, proprietatea cerută. Aşadar, trebuie să verificăm doar perechi de poziţii de forma $(1,q\geq 2)$ sau $(p\leq N-1,N)$. Pentru aceasta vom calcula sumele prefix ($SP(0)=0$; $SP(i)=SP(i-1)+h(o(i))$) şi sumele sufix ($SS(N+1)=0$; $SS(i)=SS(i+1)+h(o(i))$) şi vom verifica că acestea ($SP(q\geq 2)$ şi $SS(p\leq N-1)$) au valori în intervalul dorit.

Observăm că euristica prezentată nu ţine cont de valorile propriu-zise ale înălţimilor şi de limitele date. Ea încearcă să menţină sumele prefix cât mai aproape de media aritmetică a tuturor numerelor. Aşadar, problema poate fi generalizată în felul următor: Se dau N înălţimi cu valori în intervalul $[X,Y]$. Notăm media lor aritmetică cu M. Dorim să le ordonăm în aşa fel încât media aritmetică a oricărei subsecvenţe continue de înălţimi să fie în intervalul $[(1-f)\cdot M, (1+f)\cdot M]$, unde f este o fracţiune din M (în principiu, $0\leq f\leq 1$, dar nu este obligatoriu).

Problema 2-3. Număr maxim de interschimbări la heapsort (ACM ICPC NEERC 2004)

Algoritmul de sortare HeapSort a unei permutări cu N elemente funcţionează în felul următor. În prima etapă se construieşte un max-heap, adică un heap cu proprietatea că valoarea din orice nod este mai mare decât valorile din fiii săi. Un heap este codificat sub forma unui vector $a(1), ..., a(N)$ în felul următor. Poziţia 1 corespunde rădăcinii. Poziţiile $2\cdot i$ şi $2\cdot i+1$ corespund celor doi fii ai nodului i (dacă $2\cdot i>N$ sau $2\cdot i+1>N$, atunci fiul respectiv nu există). $a(i)$ reprezintă valoarea asociată nodului i.

În a doua etapă se extrage în mod repetat (de N ori) elementul din rădăcina heap-ului ($a(1)$). Acest element este adăugat în şirul sortat pe poziţia N, iar în heap, în locul său, este pus elementul $a(N)$. După această operaţie, N (dimensiunea heap-ului) este decrementat, iar proprietatea de heap este refăcută în modul următor. Dacă $a(1)$ este mai mic decât vreunul din fiii săi, $a(1)$ este interschimbat cu valoarea maximă a unui din cei doi fii ai rădăcinii. Fie acest fiu x. Se verifică apoi, în mod repetat, dacă $a(x)$ este mai mic decât vreunul din fiii nodului x şi, dacă da, $a(x)$ este *interschimbat* cu $a(y)$ (unde y este fiul cu valoarea maximă a nodului x), după care setăm $x=y$.

Dându-se N ($1\leq N\leq 100.000$), determinaţi un heap (ce conţine elementele $1, ..., N$), pentru care a doua parte a algoritmului HeapSort realizează un număr maxim de interschimbări.

Exemplu: N=6 => a(1)=6, a(2)=5, a(3)=3, a(4)=2, a(5)=4, a(6)=1.

Soluţie: Vom determina un heap (o secvenţă $a(1), ..., a(N)$) în mod inductiv, pornind de la secvenţa care maximizează numărul de interschimbări pentru $N-1$. Pentru $N=1$ avem o singură secvenţă ($a(1)=1$), ce generează 0 interschimbări.

Să presupunem că $a(1), ..., a(N-1)$ este secvenţa (heap-ul) care maximizează numărul de interschimbări pentru $N-1$ şi, dintre toate aceste secvenţe, are proprietatea că $a(N-1)=1$. Vom determina o secvenţă $b(1), ..., b(N)$, care maximizează numărul de interschimbări pentru un heap de dimensiune N, care va avea $b(N)=1$. După extragerea elementului din rădăcină, valoarea $b(1)$ va fi setată la 1 (iar heap-ul va avea $N-1$ elemente). Vrem acum ca elementul 1 să coboare cât mai mult în heap (până pe ultimul nivel din heap). De asemenea, vrem ca heap-ul obţinut după coborârea elementului 1 să fie $a(1), ..., a(N-1)$ (unde $a(N-1)=1$).

Aşadar, heap-ul $b(1), ..., b(N)$ se obţine după cum urmează. *Iniţializăm* $b(i)=a(i)$ ($1\leq i\leq N-1$) (momentam, lăsăm $b(N)$ nesetat). Determinăm apoi nodurile de pe drumul de la nodul $N-1$

până la rădăcină. Fie aceste noduri *v(1)=N-1, v(2)=v(1)/2, ..., v(i)=v(i-1)/2, ..., v(k)=1* (împărţirile se efectuează păstrându-se partea întreagă inferioară). Vom parcurge cu un contor *j* (*j=1,...,k-1*) şi vom seta *b(v(j))=b(v(j+1))* (practic, coborâm în heap toate elementele de pe drumul de la poziţia *N-1* la rădăcina heap-ului, suprascriind valoarea de pe poziţia *N-1*).

După această coborâre, setăm *b(1)=N* şi *b(N)=1*. Astfel, în timp logaritmic (*k=O(log(N))*), am obţinut heap-ul de dimensiune *N* care maximizează numărul de interschimbări, pornind de la heap-ul de dimensiune *N-1*. Un algoritm de complexitate *O(N·log(N))* este uşor de implementat (pornim de la un heap cu un singur element şi, pentru fiecare pas *i=2,...,N*:

(1) coborâm în jos în heap valorile din nodurile de pe drumul de la rădăcină până la nodul *i-1*;

(2) punem valoarea *i* în rădăcină;

(3) mărim dimensiunea heap-ului cu *1* element şi punem valoarea *1* în nodul *i*.

O altă soluţie este următoarea. Să considerăm că valorile de pe poziţiile *1, ..., N* ale heap-ului dorit sunt *x(1), ..., x(N)*. Ştim că *x(1)=N* şi dorim ca *x(N)=1*. În mod similar cu soluţia precedentă, vom dori ca, după fiecare extragere a elementului maxim din heap, pe ultima poziţie din heap să ajungă elementul *1*. Vom iniţializa un vector *b(i)=i* şi un graf orientat *G* cu *N* noduri şi fără nicio muchie. Apoi, pentru *i* de la *N* către *2*, efectuăm următorii paşi:

(1) eliminăm elementul *b(1)* din heap şi punem în locul său elementul *b(i)*; ştim că acum *b(1)* conţine valoarea *1*, pe care dorim să o ducem pe poziţia *b(i-1)*

(2) fie poziţiile *poz(1)=b(1), poz(2), ..., poz(K)=b(i-1)* corespunzătoare, în ordine, poziţiilor din heap de pe drumul de la rădăcină (poziţia *1*) până la ultima poziţie din heap (poziţia *i-1*)

(3) pentru fiecare indice *j=2, ..., K*, introducem muchie orientată în graful *G* de la nodul *b(poz(j))* către *b(poz'(j))*, unde *poz'(j)* este „fratele” lui *poz(j)* (dacă există) ; dacă *poz(j)* este par, atunci *poz'(j)=poz(j)+1* (altfel, *poz'(j)=poz(j)-1*).

(4) interschimbăm, pe rând, elementele de pe poziţiile *poz(j)* şi *poz(j+1)* în vectorul *b* (în ordine, de la *j=1* la *j=K-1*)

Graful *G* este un graf de ordine parţială. Dacă avem muchia orientată *i->j* în *G*, atunci trebuie ca *x(i)* să fie mai mare decât *x(j)*. *G* conţine *O(N·log(N))* muchii. În continuare vom sorta topologic nodurile din *G*, plasând nodul *N* la început şi nodul *1* la final. Vom considera apoi nodurile *i* ale lui *G* în ordinea din sortarea topologică şi vom seta *x(i)=j*, unde *j* este poziţia lui *i* în sortarea topologică ($1 \le j \le N$). În felul acesta am obţinut valorile de pe fiecare poziţie a heap-ului căutat.

Problema 2-4. Tije (Stelele Informaticii 2007)

Se consideră *N+1* tije, numerotate de la *1* la *N+1* ($1 \le N \le 100$). Tijele *1, ..., N* conţin fiecare câte *N* bile. Bilele de pe tija *i* au toate culoarea *i*. Tija *N+1* este goală. La orice moment dat, puteţi efectua mutări de tipul următor: se ia o bilă din vârful unei tije sursă şi se amplasează în vârful unei tije destinaţie, cu condiţia ca tija sursă să conţină cel puţin o bilă înaintea mutării, iar tija destinaţie să conţină cel mult *N* bile după efectuarea mutării.

Determinaţi o secvenţă de mutări astfel încât, în urma executării mutărilor, pe fiecare tijă de la *1* la *N* să se găsească câte *N* bile, fiecare bilă având o culoare diferită, iar tija *N+1* să fie goală.

Soluţie: Vom rezolva problema în *N-1* runde. La fiecare rundă *r* ($1 \le r \le N-1$) vom aduce *N* bile de culori diferite pe tija *r*. La începutul rundei *r* ($1 \le i \le N-1$), tijele *1,...,r-1* sunt deja în starea

finală, iar primele $r-1$ bile (cele din vârf) de pe tijele i ($r \le i \le N$) sunt de culori diferite de la 1 la $r-1$. Următoarele bile de pe fiecare tijă i ($r \le i \le N$) sunt de culoarea i.

Vom începe prin a muta r bile de pe tija N pe tija $N+1$. Obţinem, astfel, culorile $1,...,r-1$ şi N pe tija $N+1$. Vom parcurge apoi, în ordine descrescătoare, tijele de la $i=N-1$ până la $i=r$. Vom muta primele $r-1$ bile de pe tija i pe tija $i+1$ apoi vom muta bila rămasă în vărful tijei i (ce are culoarea i) pe tija $N+1$. Astfel, am obţinut bile de toate culorile pe tija $N+1$. Pe tija r mai avem $N-r$ bile de culoarea r. Vom muta câte o bilă de pe tija r pe fiecare din tijele $r+1, ..., N$. Astfel, tija r devine goală. În acest moment, putem muta toate bilele de pe tija $N+1$ pe tija r. Acum am ajuns la sfârşitul rundei. Primele r tije conţin bile din toate culorile, tijele $r+1, ..., N$ au primele r bile (cele din vârf) de culori $1,...,r$ (într-o ordine oarecare), iar următoarele $N-r$ bile de culoarea i (unde $r+1 \le i \le N$ este numărul tijei).

În total se efectuează $O(N^3)$ mutări.

Problema 2-5. Tăierea unui poligon convex tricolorat (TIMUS)

Se dă un poligon convex cu N ($3 \le N \le 1.000$) vârfuri. Fiecare vârf este colorat într-una din culorile *R*(oşu), *G*(alben) sau *V*(erde). Există cel puţin un vârf colorat în fiecare din cele 3 culori. Trasaţi $N-2$ diagonale care să nu se intersecteze astfel încât triunghiurile obţinute (determinate de diagonale şi laturile poligonului) să aibă fiecare câte un vârf din fiecare culoare.

Soluţie: Numărăm câte vârfuri sunt colorate în fiecare culoare C ($num(C)$). Căt timp poligonul are mai mult de 3 vârfuri, alegem un vârf v colorat într-o culoare C astfel încât $num(C)>1$ şi cele două vârfuri vecine cu vârful v (situate înainte şi după v pe conturul poligonului) să aibă culori diferite (diferite între ele şi diferite de culoarea lui v). Vom trasa diagonala determinată de cele două vârfuri situate înainte şi după v pe conturul poligonului (vârfurile a şi b). După aceasta, eliminăm vârful v de pe conturul poligonului şi vom considera diagonala proaspăt trasată (a,b) ca fiind o latură a poligonului, care are acum cu un vârf mai puţin. După eliminarea lui v decrementăm cu 1 valoarea $num(C)$ (unde C este culoarea vârfului v).

Acest algoritm determină o împărţire corectă sau nu găseşte o astfel de colorare (în niciun caz nu determină o împărţire invalidă în triunghiuri). Să analizăm ce ar putea să nu meargă bine. Să presupunem că, la un pas, toate culorile C au $num(C)=1$. În acest caz, poligonul mai are doar 3 vârfuri colorate diferit, şi cum toate triunghiurile anterioare au vâfurile colorate diferit, am găsit o soluţie corectă. Aşadar, singura problemă ce ar putea apărea ar consta în faptul că, la un moment dat, s-ar putea să nu existe niciun vârf v ai cărui vecini să fie coloraţi diferit între ei şi diferit de v.

Algoritmul ar putea fi îmbunătăţit prin folosirea unor euristici pentru alegerea nodului v care să fie eliminat (atunci când pot fi eliminate mai multe astfel de noduri). De exemplu, dintre toate nodurile ar putea fi eliminat acel nod v de culoare C astfel încât:

(1) $num(C)$ este cel mai mare dintre toţi candidaţii ; sau

(2) prin eliminarea sa, determină formarea celui mai mare număr de triunghiuri formate din 3 vârfuri consecutive pe poligon, cu toate cele 3 vârfuri colorate diferit.

Algoritmul poate fi implementat cu complexitatea $O(N^2)$.

Problema 2-6. Sortarea cheilor (Olimpiada de Informatică a Europei Centrale, 2005)

Se dă un tabel ce conţine multe rânduri şi C coloane. Valorile de pe fircare coloană sunt numerice. Asupra acestui tabel se pot efectua operaţii de tipul $Sort(k)$ ($1 \le k \le C$), care au

următorul efect: sortează rândurile tabelului crescător, în funcţie de valoarea de pe coloana *k*; dacă două rânduri au aceeaşi valoare pe coloana *k*, atunci, în urma sortării, ordinea lor relativă nu se schimbă (adică rândul aflat mai sus în tabel rămâne deasupra celui aflat mai jos).

Două secvenţe de operaţii de sortare sunt echivalente dacă produc acelaşi efect asupra tabelului, indiferent de valorile conţinute în tabel (să presupunem că $T_1(T)$ este tabelul obţinut în urma aplicării primei secvenţe de operaţii asupra unui tabel iniţial *T*, iar $T_2(T)$ este tabelul obţinut în urma aplicării celei de-a doua secvenţe asupra aceluiaşi tabel *T*; trebuie să avem $T_1(T)=T_2(T)$, oricare ar fi tabelul iniţial *T*).

Dându-se o secvenţă de operaţii de sortare, determinaţi cea mai scurtă secvenţă echivalentă cu aceasta (care conţine un număr minim de operaţii).

Exemplu: C=4, **secvenţa=Sort(1) ; Sort(2); Sort(1); Sort(2); Sort(3); Sort(3)**. Cea mai scurtă secvenţă echivalentă este: **Sort(1); Sort(2); Sort(3)**.

Soluţie: Rezolvarea problemei se bazează pe următoarea observaţie. Dacă avem o secvenţă *Sort(k); X; Sort(k)* (unde *X* este o secvenţă oarecare de operaţii), aceasta este echivalentă cu secvenţa *X; Sort(k)*. Aşadar, din secvenţa dată trebuie să păstrăm doar ultima operaţie *Sort(k)* pentru fiecare coloană *k* pentru care apare o astfel de operaţie.

Vom menţine secvenţa rezultat sub forma unei liste dublu înlănţuite (iniţial vidă). Apoi vom traversa secvenţa iniţială de la stânga la dreapta şi, pentru fiecare coloană *k* ($1 \le k \le C$), vom reţine un pointer *p(k)* către cea mai recentă apariţie a sa în cadrul secvenţei. Când întâlnim o operaţie *Sort(k)*, verificăm dacă *p(k)* este setat (iniţial, *p(k)* nu este setat, $1 \le k \le C$). Dacă *p(k)* este setat, atunci ştergem din listă elementul referenţiat de *p(k)*. Ştergerea se poate realiza în timp *O(1)*. După aceasta inserăm operaţia *Sort(k)* la sfârşitul listei şi, indiferent dacă *p(k)* era setat sau nu, setăm *p(k)* la elementul din listă corespunzător lui *Sort(k)*.

La final lista va conţine cea mai scurtă secvenţă echivalentă. Complexitatea acestei soluţii este *O(lungimea secvenţei + C)*.

O altă variantă, de complexitate *O(lungimea secvenţei+C·log(C))* constă în a reţine, pentru fiecare coloană *k*, poziţia din secvenţă pe care apare ultima operaţie *Sort(k)*. La sfârşit sortăm coloanele crescător după această poziţie (ignorându-le pe cele care nu au nicio apariţie) şi afişăm operaţiile *Sort(*)* din secvenţa echivalentă în ordinea sortată a coloanelor.

Problema 2-7. Mesaj (Lotul Naţional de Informatică, România, 2005)

Se dă un şir de caractere de lungime *L*. Acest şir conţine un mesaj ascuns şi a fost obţinut prin concatenarea cuvintelor care formau mesajul şi apoi inserarea unor caractere întâmplătoare la poziţii întâmplătoare în şir. *N* lexicografi s-au decis să descifreze mesajul. În acest scop, lexicografii vin pe rând (în ordine, de la *1* la *N*) şi fiecare lexicograf adaugă un cuvânt la un dicţionar. Iniţial dicţionarul este vid. După ce a adăugat cuvântul, fiecare lexicograf încearcă să descopere mesajul ascuns în şir. În acest scop, lexicograful partiţionează şirul într-un număr maxim de subsecvenţe, astfel încât fiecare subsecvenţă să conţină ca subşir unul dintre cuvintele existente în dicţionar la momentul respectiv. Deci numărul maxim de subsecvenţe în care a fost partiţionat şirul reprezintă de fapt numărul de cuvinte din mesaj identificate de lexicograf (cuvintele din mesaj nu sunt în mod necesar distincte). Aflaţi pentru fiecare lexicograf numărul de cuvinte din mesajul descifrat de el.

Soluţie: Imediat după citirea şirului se va calcula o matrice *First(i,c)*=cea mai mică poziţie *j* mai mare sau egală cu *i* pentru care al *j*-lea caracter din şir este egal cu *c* (*First(i,c)=i*, dacă pe poziţia *i* din şir se află caracterul *c*; altfel, *First(i,c)=First(i+1,c)*).

Cu această matrice calculată, pentru un dicţionar posibil putem aplica o soluţie greedy: se alege cuvântul pentru care segmentul în care se află ca subşir este minim ca lungime. Putem "sări" din caracter în caracter folosind matricea calculată mai sus, obţinând o complexitate $O(L \cdot S)$, unde *S* este suma lungimilor cuvintelor de până acum.

Această complexitate este mult prea mare (deoarece ea apare la fiecare cuvânt). O îmbunătăţire constă din folosirea unui *trie* în care să inserăm cuvintele. În cadrul greedy-ului, vom menţine o mulţime de noduri *M* din trie, pentru care prefixul corespunzător lor a fost deja descoperit. Când parcurgem un caracter nou *c*, pentru acele noduri ale trie-ului din *M* care au o muchie etichetată cu *c* către unul din fii *f*, introducem în *M* fiul *f*. Când am introdus în *M* un nod corespunzător unui cuvânt existent în dicţionar, atunci am găsit un nou cuvânt. După aceasta, resetăm mulţimea *M* (iniţial, ea va conţine doar rădăcina trie-ului). Totuşi, complexitatea totală nu este îmbunătăţită.

O îmbunătăţire reală este următoarea. Vom calcula pe parcurs un vector *J[i]* = cel mai mic *k* astfel încât subsecventa şirului aflată între poziţiile *i* şi *k* (inclusiv) conţine un cuvânt dat (până acum) ca subşir (sau *k=i-1* altfel). Având calculat acest vector pentru cuvintele precedente, îl putem reactualiza în complexitate $O(L \cdot x)$, unde *x* este lungimea cuvântului curent, folosind matricea *First* (considerăm fiecare poziţie de început *i* şi, folosind matricea *First(*,*)*, găsim în timp $O(x)$ cea mai mică poziţie *k* cu proprietatea că noul cuvânt apare în întervalul *[i,k]*; apoi setăm *J[i]=min{J[i], k}* sau, dacă *J[i]<i*, atunci *J[i]=k*). Cu vectorul *J* calculat, putem aplica greedy-ul în complexitatea $O(r)$, unde *r* este răspunsul. Începem cu *poz=1* (poziţiile şirului sunt numerotate de la *1* la *L*) şi *r=0*. Cât timp (*poz≤L* şi *J[poz]≥poz*): *(1) r=r+1; (2) poz=J[poz]+1*. Complexitatea totală devine $O(L \cdot S)$.

Problema 2-8. Monezi (Olimpiada Baltică de Informatică, 2006)

Se dau *N* (*1≤N≤500.000*) tipuri de monezi. Pentru fiecare tip *i* (*1≤i≤N*) se cunoaşte valoarea monezii *V(i)* (*0≤V(i)≤1.000.000.000*), precum şi dacă deţineţi vreo monedă de tipul respectiv (*T(i)=1*) sau nu (*T(i)=0*). Valorile monezilor sunt date în ordine crescătoare şi se ştie că *V(1)=1*.

Determinaţi o valoare minimă *S≤K* (*1≤K≤1.000.000.000*), astfel încât mulţimea *M* generată de următorul algoritm să conţină un număr maxim de tipuri de monezi *i* pentru care *T(i)=0*:

(1) M={};
(2) cât timp *S>0* execută:
(2.1) alege cea mai mare monedă *i* cu *V(i)≤S*;
(2.2) adaugă moneda *i* la mulţimea *M*;
(2.3) S=S-V(i)

Soluţie: Vom adăuga o monedă fictivă *N+1*, cu $V(N+1)=+\infty$ şi *T(N+1)=1*. Vom parcurge şirul monezilor în ordine crescătoare, menţinând pe parcurs suma *S* dorită. Să presupunem că am ajuns la moneda *i* (*1≤i≤N*). Dacă *S+V(i)>K*, atunci algoritmul se termină. Dacă *T(i)=1*, atunci trecem la moneda următoare; altfel, vom verifica dacă putem modifica suma *S* astfel încât să fie generată o monedă de tipul *i* de către algoritm. Dacă *S+V(i)<V(i+1)*, atunci *V(i)* este prima monedă aleasă de algoritm pentru suma *S'=S+V(i)*; după ce este aleasă această monedă, suma devine *S* şi, în continuare, vor fi alese monezile corespunzătoare sumei *S*.

Aşadar, dacă $S+V(i)<V(i+1)$, atunci setăm $S=S+V(i)$, adăugăm tipul i la mulţimea de monezi ce vor fi alese şi trecem la moneda următoare $i+1$. Suma finală S maximizează numărul de monezi alese de algoritm şi este cea mai mică sumă cu această proprietate.

Problema 2-9. Şirag (Lotul Naţional de Informatică, România 2008)

Pentru a intra în Cartea Recordurilor, locuitorii din Văscăuţi vor face un şirag de mărgele foarte foarte lung. În acest scop ei au cumpărat mărgele de K ($2 \le K \le 100.000$) culori (pentru fiecare culoare i fiind cunoscut numărul $a(i)$ de mărgele cumpărate; $1 \le a(i) \le 10^9$). Locuitorii din Văscăuţi consideră că şiragul este frumos dacă oricare secvenţă de P ($2 \le P \le K$) mărgele consecutive din şirag ($2 \le P \le K$) nu conţine două mărgele de aceeaşi culoare. Determinaţi lungimea maximă a unui şirag frumos care se poate construi cu mărgelele cumpărate.

Soluţie: O serie de soluţii oferite de d-na prof. Emanuela Cerchez sunt următoarele.

1) Vom sorta descrescător vectorul a. Considerăm primele p valori din şirul a şi le vom scădea cu $a(p)$ (ceea ce ar însemna că vom plasa mărgelele $1, 2, ..., p$ de $a(p)$ ori. Resortăm vectorul a şi repetăm operaţia.
Complexitate: $O(K^2 \cdot log(K))$

2) Observăm că după aplicarea unui pas, tabloul a este divizat în două subsecvenţe descrescătoare. Pentru sortare este suficient sa interclasăm cele două secvenţe. Complexitatea va fi în acest caz $O(K^2)$.

3) O altă soluţie este să organizăm valorile $a(1), a(2), ..., a(K)$ ca un heap şi la fiecare pas să extragem de p ori maximul, reducem elementele şi le reinserăm în heap.
Complexitate: $O(K \cdot P \cdot log(K))$.

4) Complexitate: $Sortare+O(P \cdot log(S))$. (unde $S=a(1)+...+a(K)$)
Sortăm vectorul a descrescător. Calculăm în vectorul SP şirul sumelor parţiale: $SP(i)=a(i)+a(i+1)+...+a(K)$ ($SP(K+1)=0$ şi $SP(1 \le j \le K)=a(j)+SP(j+1)$). Vom căuta binar (în intervalul $[0, (SP(1)\ div\ P)+2]$) numărul maxim de secvenţe de lungime P pe care le putem forma. Funcţia *good(x)* verifică dacă putem construi x secvenţe de lungime P formate din valori distincte:

```
int good(long long  x)
{for (i=0; i<p; i++)
    {if (x*(p - i) > SP[i]) return -1;
     if (a[i] <= x) return i;}
 return p;}
```

Funcţia returnează valoarea -1 dacă nu este posibil, respectiv o valoare ≥ 0 (poziţia) în caz contrar. Lungimea şiragului se determină astfel: $rez \cdot p+rest$, unde: $rest=poz+SP[poz]-rez \cdot (p-poz)$ ($poz=good(rez)$).

O altă soluţie bazată tot pe căutarea binară a numărului de secvenţe de lungime P care să fie formate din valori distincte este următoarea. Căutăm valoarea *rez* în acelaşi interval ca şi înainte. Testul de fezabilitate (care verifică dacă se pot forma x astfel de secvenţe) este următorul. Iniţializăm 2 contoare Q şi R cu 0. Apoi parcurgem toate tipurile i de mărgele (toate culorile). Dacă $a(i)>x$, setăm $Q=Q+x$; altfel, $Q=Q+a(i)$; dacă, la un moment dat, avem $Q \ge x$, atunci setăm $R=R+1$ şi $Q=Q\ mod\ x$. Dacă privim cele x secvenţe ca fiind x rânduri ale unei matrici cu x linii şi P coloane, R reprezintă numărul de coloane pe care am reuşit să le completăm în aşa fel încât fiecare din cele x linii să conţină elemente de valori (culori) diferite. Dacă $R \ge P$, atunci putem forma x secvenţe de câte P mărgele de culori

diferite => x este o valoare fezabilă şi vom căuta în continuare un x mai mare. Altfel, x nu este fezabil şi vom căuta în continuare un x mai mic.

La sfârşitul căutării binare am obţinut un şir de lungime $rez \cdot P$. Mai trebuie să calculăm restul (câte mărgele mai pot fi adăugate la sfârşit, între 0 şi $P-1$). Vom iniţializa contoarele R, Q şi $nextra$ la 0 şi apoi vom parcurge cele K tipuri (culori) de mărgele ($i=1,...,K$). Dacă $a(i)>rez$, atunci setăm $R=R+rez$ şi $Q=Q+1$; altfel efectuăm următoarele acţiuni:

(1) $R=R+a(i)$;

(2) dacă $R>rez \cdot P$ atunci $\{R=R-1;\ Q=Q+1\}$; altfel $nextra=nextra+1$.

Restul va fi egal cu $rest=Q+min\{R-rez \cdot P,\ nextra\}$. Aşadar, lungimea maximă este $rez \cdot P+rest$.

Problema 2-10. Număr minim de greutăţi

Determinaţi o mulţime W cu număr minim de greutăţi întregi, care are proprietatea că orice greutate întreagă între 1 şi N ($1 \le N \le 10^{100}$) poate fi cântărită folosind fiecare greutate din W cel mult o dată şi având la dispoziţie un cântar cu două talere. Considerăm două cazuri:

(1) greutăţile se pun doar pe un taler şi greutatea de cântărit se pune pe celălalt taler;

(2) greutăţile din W pot fi puse pe ambele talere (inclusiv lângă greutatea de cântărit).

Soluţie: Pentru cazul *(1)*, mulţimea W constă din puterile lui 2 mai mici sau egale cu N: $W=\{2^i | 0 \le i \le$ *parte întreagă inferioară*$(log_2(N))\}$. Pentru a cântări o greutate X determinăm reprezentarea sa în baza 2. Pentru fiecare bit i ($0 \le i \le log_2(N)$) de 1, punem greutatea 2^i pe talerul din stânga. Apoi punem greutatea X pe talerul din dreapta.

Pentru cazul *(2)* mulţimea W constă din puterile lui 3 mai mici sau egale cu N, plus cea mai mică putere a lui 3 mai mare sau egală cu N: $W=\{3^i | 0 \le i \le$ *parte întreagă superioară*$(log_3(N))\}$. Când avem de cântărit o greutate X procedăm după cum urmează. Determinăm reprezentarea în baza 3 a lui X. Fie $Q(i)$ puterea la care apare 3^i în reprezentarea lui X ($Q(i)=0$, 1 sau 2). Parcurgem poziţiile i crescător, începând de la 0, până la valoarea maximă posibilă (parte întreagă superioară din $log_3(N)$). Vom considera că greutatea X este amplasată pe talerul din stânga. Dacă $Q(i)=1$, atunci punem greutatea 3^i pe talerul din dreapta (cel opus lui X) şi setăm $Q(i)=0$. Dacă $Q(i)=2$, atunci punem greutatea 3^i pe talerul din stânga (lângă X), după care setăm $Q(i)=0$ şi $Q(i+1)=Q(i+1)+1$. Dacă $Q(i)=0$, atunci trecem la poziţia următoare. Observăm că, în cadrul algoritmului, putem ajunge şi cu Q(i)=3 (deoarece realizăm şi incrementări cu 1). Dacă ajungem la o poziţie i unde $Q(i)=3$, atunci setăm $Q(i)=0$ şi $Q(i+1)=Q(i+1)+1$; apoi trecem mai departe. Observăm că în felul acesta, în cel mai rău caz, este posibil să folosim o greutate 3^i unde 3^i este cea mai mică putere a lui 3 mai mare sau egală decât X.

Problema 2-11. Traverse (Lotul Naţional de Informatică, România 2000)

2 locomotive se află la capete opuse ale unei şine ce constă din N ($2 \le N \le 100.000$) poziţii (prima locomotivă pe poziţia 1, cealaltă pe poziţia N). Pe fiecare poziţie i se află sau nu o traversă ($1 \le i \le N$). O locomotivă se poate deplasa de la poziţia curentă $i \ge K$ (respectiv $i \le N-K+1$) la poziţia următoare ($i+1$ pentru prima locomotivă şi $i-1$ pentru a doua), numai dacă pe poziţia următoare se află o traversă sau dacă printre ultimele K poziţii traversate (inclusiv cea curentă) se află cel puţin o poziţie care conţine o traversă. Dacă nici poziţia următoare şi nici ultimele K ($1 \le K \le N$) poziţii parcurse nu conţin nicio traversă, atunci mecanicul locomotivei va trebui să demonteze o traversă de pe o poziţie parcursă anterior şi să o monteze pe poziţia următoare. După această mutare, locomotiva se poate muta pe poziţia următoare; dacă nu

există nicio traversă pe poziţiile parcurse anterior şi locomotiva se află în situaţia de a nu putea realiza deplasarea pe poziţia următoare, atunci locomotiva rămâne blocată pe poziţia curentă.

Cele *2* locomotive doresc să ajungă una lângă cealaltă, pe poziţii consecutive (prima pe o poziţie *i*, iar a doua pe poziţia *i+1*; $1 \le i \le N-1$). Determinaţi numărul minim de mutări ce trebuie efectuate pentru ca cele *2* locomotive să ajungă pe poziţii alăturate.

Soluţie: Vom calcula pentru prima locomotivă valorile *Tleft(i)*=numărul minim de mutări ce trebuie realizate pentru ca locomotiva să ajungă de pe poziţia *1* pe poziţia *i*. Avem *Tleft(0)=0*. Vom parcurge valorile lui *i* ($1 \le i \le N$) în ordine crescătoare şi vom menţine două contoare: *nlipsa*=numărul de poziţii consecutive fără traverse parcurse de locomotivă de la ultima poziţie cu traversă parcursă, şi *ntrav*=numărul de traverse peste care a trecut locomotiva. Iniţial, *nlipsa=ntrav=0*.

Dacă pe poziţia *i* se află o traversă, atunci setăm *Tleft(i)=Tleft(i-1)*, *nlipsa=0* şi *ntrav=ntrav+1*. Dacă pe poziţia *i* nu se află o traversă şi $nlipsa < K$, atunci setăm *Tleft(i)=Tleft(i-1)* şi *nlipsa=nlipsa+1*. Dacă pe poziţia *i* nu se află traversă şi *nlipsa=K*, atunci verificăm dacă $ntrav \ge 1$. Dacă $ntrav \ge 1$, atunci vom muta o traversă anterioară pe poziţia *i* şi apoi vom seta *Tleft(i)=Tleft(i-1)+1* şi *nlipsa=0*. Vom calcula, în mod similar, nişte valori *Tright*, parcurgând poziţiile descrescător, de la *N* către *1*. Avem *Tright(N+1)=0*. Pentru $1 \le i \le N$ calculăm *Tright(i)* pe baza lui *Tright(i+1)*, considerând aceleaşi cazuri ca şi în cazul lui *Tleft(i)* (care a fost calculat pe baza lui *Tleft(i-1)*). Vom calcula apoi $Vmin=min\{Tleft(i)+ Tright(i+1) | 1 \le i \le N-1\}$, reprezentând numărul total minim de mutări necesare. Complexitatea algoritmului este *O(N)*.

Problema 2-12. Cărţi (Olimpiada Baltică de Informatică 2005)

Aveţi la dispoziţie *N* cărţi de joc ($1 \le N \le 100.000$). Fiecare carte *i* ($1 \le i \le N$) are două feţe şi pe fiecare din ele este scris un număr: *a(i)* şi *b(i)* ($-1000 \le a(i),b(i) \le 1000$). Trebuie să aşezaţi toate cărţile pe masă, cu una din feţe în sus. Pe *K* ($0 \le K \le N$) dintre cărţi trebuie să schimbaţi semnul numărului de pe faţa de sus. Fie *S* suma numerelor de pe feţele de sus ale celor *N* cărţi (considerând numerele cu semn schimbat pentru cele *K* cărţi alese). Alegeţi câte o faţă pentru fiecare carte şi alegţi cele *K* cărţi căror să le schimbaţi semnul, astfel încât *S* să fie minim.

Soluţie: Dacă o carte *i* este aleasă pentru a-i schimba semnul feţei de sus, atunci vom pune pe faţa de sus valoarea maximă *cmax(i)=max{a(i),b(i)}*. Dacă nu este aleasă, pe faţa de sus ea ar trebui să aibă valoarea minimă *cmin(i)=min{a(i), b(i)}*. O primă soluţie constă din construirea unui graf bipartit care conţine cele *N* cărţi în partea stângă şi *2* noduri (*X* şi *Y*) în partea dreaptă. O muchie de la o carte *i* la nodul *X* are capacitatea *1* şi costul *–cmax(i)*, iar muchia *(i,Y)* are capacitate *1* şi costul *cmin(i)*.

Mai introducem două noduri speciale *S* şi *D*. Ducem muchii de capacitate *1* (şi cost *0*) de la *S* la fiecare carte *i*; introducem şi muchia *(X,D)*, de capacitate *K* şi muchia *(Y,D)*, de capacitate *N-K*.

Vom calcula un flux maxim de cost minim în acest graf. La final, fluxul va fi egal cu *N*, iar costul minim va reprezenta suma minimă posibilă *S*. Totuşi, complexitatea acestui algoritm este prea mare.

O a doua soluţie, mult mai simplă, este următoarea. Vom presupune iniţial că toate cărţile contribuie la sumă cu semnul neschimbat. În acest caz, avem suma *S'*=suma valorilor *cmin(i)*

($1 \le i \le N$). Vom alege apoi acele K cărţi pentru care diferenţa dintre noua lor contribuţie şi vechea lor contribuţie este minimă. Astfel, vom sorta crescător cărţile după valoarea $dif(i)=-cmax(i)-cmin(i)$ şi le vom alege pe primele K din această ordine. Fie $Sdif$ suma celor mai mici K valori $dif(*)$. Suma minimă S este egală cu $S'+Sdif$.

Problema 2-13. Acoperire cu număr minim de intervale

Se dau N ($1 \le N \le 100.000$) intervale închise. Intervalul i ($1 \le i \le N$) este $[A(i), B(i)]$. Se mai dă şi un interval special $[U,V]$. Determinaţi $K \ge 1$ submulţimi disjuncte ale mulţimii de intervale $\{1,\dots,N\}$, astfel încât reuniunea intervalelor din fiecare submulţime să includă intervalul $[U,V]$.

Soluţie: Vom trata întâi cazul $K=1$. Vom sorta intervalele după capătul lor stânga şi le vom parcurge în această ordine. Vom adăuga şi un interval fictiv $[V+1, \infty]$.

Pe durata parcurgerii vom menţine o variabilă *xc*, pe care o iniţializăm la U. De asemenea, vom menţine două variabile, *xcand* şi *icand*, pe care le iniţializăm la U, respectiv 0.

Să presupunem că, în cadrul parcurgerii, am ajuns la intervalul i. Dacă $A(i)>xc$, atunci verificăm dacă $icand \ne 0$; dacă $icand=0$, atunci algoritmul se termină; dacă $icand \ne 0$, atunci adăugăm intervalul *icand* la submulţime, setăm $xc=xcand$, după care resetăm $icand=0$ – dacă am ajuns cu $xc \ge V$, atunci algoritmul se termină. După aceste procesări, verificăm dacă $A(i) \le xc$ şi $B(i)>xcand$; dacă da, atunci setăm $icand=i$ şi $xcand=B(i)$.

Dacă la final avem $xc \ge V$, atunci am găsit o submulţime cu număr minim de intervale a căror reuniuni include intervalul $[U,V]$. Observăm că algoritmul a funcţionat după cum urmează: de fiecare dată a ales intervalul care are capătul dreapta cel mai mare şi care are o intersecţie nevidă cu ultimul interval adăugat la submulţime. Complexitatea algoritmului este $O(N \cdot log(N))$ de la sortare, plus $O(N)$ de la parcurgerea intervalelor.

Putem generaliza această soluţie pentru $K>1$. Vom folosi o structură de date *Dxc* care va conţine până unde a ajuns reuniunea intervalelor din fiecare submulţime (în *Dxc* vom reţine perechi *(xc, idx)*; unde $1 \le i \le K$). Vom folosi, de asemenea, o structură *Dxcand* care va conţine intervalele candidate (cel mult K) cu cele mai mari capete dreapta.

Vom iniţializa *Dxc* prin inserarea perechilor $(xc=U, idx=i)$ ($1 \le i \le K$). *Dxcand* va fi, iniţial, vidă. *Dxc* oferă funcţiile:

- *getMin()*, care va întoarce perechea *(xc, idx)* cu valoare minimă a lui *xc*;
- *delMin()*, care va şterge perechea *(xc, idx)* cu *xc* minim din *Dxc*;
- *add(xc, idx)*, care adăugă perechea (xc, idx) în *Dxc*.

Dxcand oferă funcţii similare:

- *getMax()* va întoarce perechea *(xcand, icand)* cu valoare maximă a lui *xcand*;
- *delMax()* va şterge din *Dxcand* perechea *(xcand, icand)* cu valoare maximă a lui *xcand*;
- *add(xcand, icand)* adaugă perechea *(xcand, icand)* în *Dxcand*.

Vom parcurge apoi cele N intervale în aceeaşi ordine ca şi în soluţia pentru $K=1$ (în ordine crescătoare ale capetelor stânga), folosind şi de această dată intervalul fictiv $[V+1, \infty]$.

Să presupunem că în cadrul parcurgerii am ajuns la intervalul i. Cât timp $A(i)>Dxc.getMin().xc$ vom efectua următoarele acţiuni:

(1) dacă *Dxcand* este vidă, atunci algoritmul se termină;

(2) altfel, fie *(xcand, icand)=Dxcand.getMax()* şi *(xc, idx)=Dxc.getMin()*; dacă $xcand>xc$, atunci apelăm *Dxc.delMin()*, *Dxcand.delMax()*, după care adăugăm perechea *(xcand, idx)* în

Dxc, precum şi intervalul *icand* în submulţimea de intervale *S(idx)* (presupunem că avem *K* astfel de submulţimi, numerotate de la *1* la *K*).

Dacă avem *Dxc.getMin().xc≥V*, atunci algoritmul se termină. Altfel, vom considera intervalul *i* la care am ajuns drept un interval candidat. Vom introduce perechea *(B(i), i)* în *Dxcand*.

Din punct de vedere al implementării, *Dxc* şi *Dxcand* pot fi:

(1) vectori menţinuţi sortaţi;

(2) min-heap şi, respectiv, max-heap;

(3) arbori echilibraţi.

În cazul *(1)*, operaţia de găsire a minimului (maximului) durează *O(1)*, iar o operaţie de inserare/ştergere durează *O(lungimea vectorului)*. *Dxc* conţine mereu *K* elemente, deci inserările şi ştergerile vor dura *O(K)*. În cazul lui *Dxcand*, am putea ajunge cu *O(N)* elemente în vector, deci o inserare/ştergere ar putea dura *O(N)*. Dacă, însă, după fiecare inserare după care avem mai mult de *K* elemente în *Dxcand* ştergem elementul minim, pentru a rămâne cu cel mult *K* elemente, lungimea lui *Dxcand* este limitată la *K* elemente, iar operaţiile de inserare/ştergere durează *O(K)*.

În cazul min-heap-ului (max-heap-ului), găsirea minimului (maximului) se face în *O(1)*, iar inserarea unui element sau ştergerea minimului (maximului) se fac în *O(log(K))* (pentru *Dxc*), respectiv *O(log(N))* pentru *Dxcand*. În acest caz, e mai greu să limităm numărul de elemente din *Dxcand* la *K*, deoarece ar trebui să avem acces şi la perechea *(xcand, icand)* din *Dxcand* cu *xcand* minim. Pentru asta, am putea folosi un min-max-heap în locul unui max-heap.

Dacă folosim arbori echilibraţi, toate operaţiile au complexitatea *O(log(K))* pentru *Dxc* şi *O(log(N))* sau *O(log(K))* pentru *Dxcand*. Folosind arbori echilibraţi putem avea acces uşor la cel mai mic şi cel mai mare element din *Dxcand*, astfel că putem realiza ştergeri pentru a menţine numărul de elemente la cel mult *K*.

În cel mai bun caz, aşadar, complexitatea soluţiei este *O(N·log(K))*. O extensie interesantă (pentru *K=1*) este următoarea: fiecare interval *i* are o pondere *w(i)≥0* şi dorim să găsim o submulţime de intervale cu suma minimă a ponderilor, astfel încât reuniunea lor să includă intervalul *[U,V]*.

Putem rezolva această problemă după cum urmează. Vom sorta toate capetele stânga şi dreapta ale intervalelor (inclusiv ale intervalului *[U,V]),* obţinând un şir *x(1)<x(2)<...<x(Q)* (unde *Q≤2·n+2* este numărul de coordonate distincte). Fie *i* şi *j* indicii pentru care *x(i)=U* şi *x(j)=V*.

Vom construi o mulţime de puncte *P*, după cum urmează. Dacă *i=j*, *P={x(i)}*. Altfel, vom adăuga la *P* puncte având coordonate de forma: *(x(g)+x(g+1))/2* (*i≤g≤j-1*). Acum putem folosi un algoritm de programare dinamică, prezentat în [Andreica-MCBE2008], pentru a găsi o submulţime de intervale de pondere totală minimă pentru a acoperi mulţimea de puncte P. Trebuie doar să observăm că dacă fiecare punct din P este acoperit de un interval din submulţime, atunci întregul interval *[U,V]* este acoperit de un interval din submulţime. Complexitatea algoritmului de programare dinamică este *O(N·log(N))*.

Problema 2-14. Bilete la concert (Olimpiada de Informatică a Europei Centrale, 2005)

Într-o sală de concert există *N* (*1≤N≤100.000*) locuri, aşezate în linie şi numerotate de la *1* la *N*, de la stânga la dreapta. Întrucât urmează să aibă loc un concert al unei formaţii foarte populare, organizatorii concertului au scos la vânzare biletele pentru concert, pe care le vând în intervale de câte *L* (*1≤L≤M*) locuri consecutive. S-au primit *M* (*1≤M≤100.000*) cereri de

cumpărare. O cerere i ($1 \le i \le M$) specifică primul loc $P(i)$ pentru care se doresc bilete ($1 \le P(i) \le N-L+1$); astfel, cererea doreşte cumpărarea biletelor pentru locurile $P(i)$, $P(i)+1$, ..., P(i)+L-1.

Preţul cumpărării biletelor pentru L locuri consecutive, începând de la primul loc specificat, este 2 RON. Organizatorii şi-au dat seama că ar putea obţine un profit mai mare dacă, pentru unele cereri, nu ar aloca exact cele L locuri consecutive cerute (începând de la poziţia $P(i)$), ci alte L locuri consecutive. În acest caz, cei care au emis cererea ar cumpăra oricum biletele (concertul fiind foarte popular), însă nu ar fi dispuşi să plătească decât 1 RON (întrucât intervalul de L locuri consecutive nu începe de unde au dorit ei).

Primind foarte multe cereri, este posibil ca unele dintre ele să fie refuzate (întrucât ori nu pot fi satisfăcute, ori nu ar fi folositoare într-o strategie de vânzare a biletelor care aduce profitul maxim). Ştiind că biletul unui loc nu poate fi vândut decât în cadrul unui interval de L locuri consecutive asociat unei singure cereri, determinaţi profitul maxim pe care îl pot obţine organizatorii concertului.

Soluţie: În prima etapă vom sorta cererile descrescător după primul loc cerut din cadrul intervalului de L locuri consecutive (adică după valorile $P(i)$). Vom menţine o mulţime S a cererilor satisfăcute total (iniţial, S este vidă), precum şi o valoare $Pmin$, reprezentând cel mai din stânga loc ocupat până acum (iniţial, $Pmin=N+1$).

Vom parcurge cererile în ordinea sortată descrescător după $P(i)$. Când ajungem la o cerere i, dacă $P(i) \le Pmin-L$, atunci adăugăm cererea i la mulţimea S şi setăm $Pmin=P(i)$; altfel, trecem la cererea următoare. Vom nota cererile din S prin $S(1), ..., S(K)$, unde $S(1)$ este ultima cerere adăugată, iar $S(K)$ este prima cerere adăugată în S (deci, sunt numerotate în ordine inversă adăugării în S, dar în ordine crescătoare a primului loc $P(i)$ cerut; K=numărul total de cereri adăugate în S). Pentru fiecare cerere i, vom menţine um marcaj care indică dacă i face parte din S sau nu (astfel, verificarea dacă o cerere i face parte din S sau nu se realizează în $O(1)$).

În al doilea pas vom modifica mulţimea S, fără a-i micşora cardinalul. Vom sorta crescător după valoarea $P(i)$ toate cererile (inclusiv cele din S). Vom menţine un indice q care reprezintă o cerere din S ($S(q)$) care ar putea fi înlocuită de o cerere mai „eficientă”.

Iniţial, $q=1$. Vom parcurge apoi cererile, în ordinea sortată. Să presupunem că am ajuns la o cerere i. Dacă i nu este în S, atunci: cât timp $P(i) \ge P(S(q))$ şi $q<K$, incrementăm indicele q cu 1. La ieşirea din acest ciclu, dacă ($P(i)<P(S(q))$) şi (($q=1$) sau ($q>1$ şi $P(i) \ge P(S(q-1))+L$)), atunci vom efectua următoarele acţiuni:

(1) fie $FFree$=cea mai din stânga poziţie liberă, cu proprietatea că toate poziţiile $FFree$, $FFree+1, ..., P(S(q))-1$ sunt şi ele libere; dacă $q=1$, $FFree=0$; altfel, $FFree=P(S(q-1))+L$; *(2)* dacă ($P(i)<P(S(q))$) şi $(((P(i) - FFree) \bmod L) < ((P(S(q)) - FFree) \bmod L))$, atunci vom înlocui cererea $S(q)$ din S cu cererea i (practic, vom marca că cererea $S(q)$ nu mai face parte din S; apoi vom seta $S(q)=i$ şi vom marca că cererea i face parte din S).

În ultima etapă vom umple golurile dintre cererile din S, folosind cererile rămase. Au mai rămas $M-K$ cereri nesatisfăcute total. Vom calcula numărul C de cereri ce mai pot fi adăugate. Iniţializăm C cu $((P(S(1))-1) \text{ div } L)$ (*div* întorce câtul împărţirii întregia două numere). Apoi incrementăm C cu fiecare valoare $(P(S(q)) - P(S(q-1)) - L) \text{ div } L$ ($2 \le q \le K-1$). La final, mai adăugăm la C valoarea $((N-P(S(K))-L+1) \text{ div } L)$.

Profitul maxim ce poate fi obţinut este $2 \cdot K+min\{M-K, C\}$. Complexitatea algoritmului este $O(M \cdot log(M))$, pentru sortarea cererilor. Putem folosi şi o variantă a sortării prin numărare, pentru a ajunge la o complexitate $O(N+M)$ (pentru fiecare poziţie j reţinem o listă

cu toate cererile i care au $P(i)=j$; apoi concatenăm aceste liste în ordine crescătoare sau descrescătoare a poziţiei j, obţinând sortarea dorită).

Demonstraţia că algoritmul găseşte o soluţie optimă *SALG* are la bază următoarele tehnici. Presupunem că există o soluţie optimă *SOPT*. Putem arăta că *SOPT* poate fi modificată fără a-i micşora profitul total, astfel încât să ajungem la SALG. Pentru început, putem arăta că *SOPT* poate fi modificată pentru a conţine K cereri satisfăcute total. Este clar că nu poate conţine mai mult de K (deoarece K a fost calculat astfel încât să reprezinte numărul maxim de cereri satisfăcute total). Dacă *SOPT* conţine mai puţin de K cereri, atunci putem înlocui/ adăuga acele cereri X din *SALG* la *SOPT*, care nu se suprapun cu cererile satisfăcute total din *SOPT* (o cerere X din *SALG* se poate intersecta cu cel mult 2 cereri Z satisfăcute parţial în *SOPT*, care vor fi eliminate şi înlocuite de X, fără a micşora profitul total).

Apoi, dacă tot nu avem K cereri satisfăcute total în *SOPT*, înseamnă că avem o cerere X satisfăcută total în *SOPT* care intersectează 2 cereri Y satisfăcute total în *SALG*. Vom elimina cererea X şi cel mult alte 2 cereri satisfăcute parţial în *SOPT* şi le vom înlocui cu cererile din *SALG*. Astfel, vom ajunge să avem K cereri satisfăcute total în *SOPT*. Dintre toate soluţiile cu K cereri satisfăcute total, cea construită conform algoritmului descris maximizează numărul de cereri satisfăcute parţial care pot fi puse pe lângă cele K cereri satisfăcute total. Astfel, demonstraţia este completă.

Problema 2-15. Verificarea unor secvenţe de operaţii

Se dau d ($1 \le d \le 5$) secvenţe de operaţii ce formează un program. Secvenţa i constă din $m(i)$ ($1 \le m(i) \le 100.000$; $m(1) \cdot \ldots \cdot m(d) \le 10.000.000$). Operaţia j din secvenţa i ($1 \le i \le d$; $1 \le j \le m(i)$) este de forma $Add(q,x)$ şi reprezintă adunarea valorii x la variabila q.

Există în total K variabile, numerotate de la 1 la K, având valorile iniţiale $v(1), ..., v(K)$. Pentru fiecare variabilă q există o funcţie $F(q,x)$ care întoarce 1 dacă se consideră *OK* cazul în care valoarea variabilei q ar fi x, respectiv 0, dacă nu este considerată *OK*.

Cele d secvenţe de operaţii se execută în paralel. O stare a programului constă dintr-un tuplu $(op(1), ..., op(d))$ ($0 \le op(i) \le m(i)$; $1 \le i \le d$), având semnificaţia că s-au executat $op(i)$ operaţii din secvenţa de operaţii i. Pentru fiecare stare $(op(1), ..., op(d))$ vrem să determinăm numărul de variabile q pentru care $F(q,v'(q))=1$ ($v'(q)$ este valoarea curentă a variabilei q, care depinde doar de operaţiile efectuate până acum în cele d secvenţe). Vom nota acest număr prin $NV(op(1), ..., op(d))$.

Soluţie: Vom considera, pe rând, fiecare tuplu $(op(1), ..., op(d-1))$ şi vom reţine, pentru fiecare variabilă q, $vS(q, op(1), ..., op(d-1))$. Avem $vS(q, op(1)=0, ..., op(d-1)=0)=v(q)$. Pentru cazul când avem cel puţin o valoare $op(j)>0$, $vS(q, op(1), ..., op(d-1))=vS(op(1), ..., op(j-1), op(j)-1, op(j+1), ..., op(d-1))$+(dacă operaţia $op(j)$ a secvenţei j este $Add(q,x)$, atunci x; altfel, 0).

Pentru fiecare tuplu $(op(1), ..., op(d-1))$ vom considera, pe rând, valorile lui $op(d)$ (de la 0 la $m(d)$). Vom iniţializa un contor *NOK*=numărul de variabile j pentru care $F(q, vS(q, op(1), ..., op(d-1)))=1$. Fie $v'(q)=vS(q, op(1), ..., op(d-1))$. Avem $NV(op(1), ..., op(d-1), 0)=NOK$. Când trecem de la valoarea $op(d)-1$ la valoarea $op(d)$, incrementăm $v'(q)$ cu x, unde operaţia $op(d)$ a secvenţei d este $Add(q,x)$. Apoi setăm $NOK=NOK+F(q, v'(q))-F(q, vant(q))$ (unde $vant(q)$ este valoarea anterioară a variabilei q, adică $v'(q)-x$) şi $NV(op(1), ..., op(d))=NOK$.

Complexitatea acestui algorithm este $O(m(1)\cdot...\cdot m(d-1)\cdot(K+m(d)))$. Observăm că putem considera secvenţele în orice ordine. Astfel, a *d*-a secvenţă va fi cea cu număr maxim de operaţii.

O aplicaţie a acestei probleme constă în detecţia existenţei de deadlock-uri (potenţiale) în programe concurente. Să presupunem că fiecare variabilă q corespunde unui semafor care are o valoare iniţială $v(q)\geq 0$. Cele d secvenţe de operaţii reprezintă d thread-uri care se execută în paralel. O operaţie $Add(q,x)$ incrementează sau decrementează valoarea semaforului q. Dacă în urma operaţiei, valoarea semaforului q ar trebui să devină negativă, atunci thread-ul respectiv se blochează până când alt thread măreşte valoarea semaforului q suficient de mult. Un deadlock este o situaţie în care există cel puţin un thread care nu şi-a încheiat execuţia, şi niciunul din thread-urile care nu şi-au încheiat execuţia nu pot executa următoarea operaţie (deoarece ar micşora sub 0 valoarea unui semafor).

Vom construi un graf al stărilor $(op(1), ..., op(d))$. Funcţia $F(q,x)$ întoarce 1 dacă $x\geq 0$, şi -1, dacă $x<0$. Vom aplica algoritmul descris anterior şi vom marca ca fiind *invalide* toate stările $(op(1), ..., op(d))$ pentru care $NV(op(1), ..., op(d))<K$. Graful va consta numai din stările *valide*. Vom avea muchii orientate de la o stare validă $(op(1), ..., op(d))$ către fiecare din stările valide de forma $(op'(1), ..., op'(d))$, unde $op'(j)=op(j)+1$ şi $op'(i\neq j)=op(i)$ $(1\leq j\leq d)$. Vom parcurge apoi graful începând din starea $(0, 0, ..., 0)$ şi vom marca ca fiind *atinse* toate stările la care putem ajunge (vom folosi, de exemplu, o parcurge în lăţime sau în adâncime). Toate stările $(op(1), ..., op(d))$ din care nu iesie nicio muchie şi care au cel puţin o valoare $op(i)<m(i)$ $(1\leq i\leq d)$ sunt stări de deadlock. Dacă există cel puţin o stare de deadlock care să fie şi validă, şi atinsă, atunci programul prezintă posibilitatea apariţiei deadlock-urilor.

Problema 2-16. Pătrate (Lotul Naţional de Informatică, România 2007, enunţ modificat)

Pe o foaie de matematică sunt LxL $(1\leq L\leq 5.000)$ pătrăţele cu latura egală cu 1 cm. Pătrăţelele sunt evident organizate în L linii (numerotate de sus în jos de la 1 la L) şi L coloane (numerotate de la stânga la dreapta de la 1 la L). Poziţia fiecărui pătrăţel de pe foaie este caracterizată prin numărul liniei şi numărul coloanei pe care se află pătrăţelul.

Pe foaie sunt înnegrite N $(1\leq N\leq 100.000)$ pătrăţele. Trebuie să desenăm pe foaia de matematică o mulţime de pătrate care să îndeplinească următoarele condiţii:

- aria intersecţiei între oricare două pătrate din această mulţime este 0;
- oricare dintre pătratele acestei mulţimi este alcătuit doar din pătrăţele întregi;
- oricare pătrăţel negru aparţine exact unuia dintre pătratele mulţimii;
- pentru oricare dintre pătratele acestei mulţimi, dacă notăm cu S aria pătratului, atunci suprafaţa ocupată de pătrăţelele negre din interiorul respectivului pătrat aparţine intervalului $[S/k, 4S/k)$, unde k $(5\leq k\leq 10)$ este un număr natural nenul dat.

Se garantează că $N<L^2/k$. Determinaţi o mulţime de pătrate care să respecte condiţiile de mai sus.

Soluţie: Pentru început vom considera pătratul de latură $x=L$, având colţul din stânga-sus la $(1,1)$, care cuprinde toate pătrăţelele negre şi în care aria ocupată de pătrăţelele negre este mai mică decât S/k, unde S este aria pătratului de latură x (adică x^2).

Împărţim acest pătrat în patru pătrate egale disjuncte de latură $x/2$ (prin cele două linii mediane) şi în aceste 4 pătrate aria punctelor negre nu va depăşi $4\cdot S/k$ (deoarece aria celor 4 pătrate este de patru ori mai mică decît aria pătratului iniţial). Pentru fiecare dintre aceste patru pătrate avem unul din următoarele 3 cazuri:

- el nu conţine nici un pătrăţel negru; în acest caz, pătratul respectiv nu mai prezintă niciun interes
- el conţine un număr de pătrăţele negre a căror arie însumată se încadrează între limitele impuse prin enunţ: în acest caz el va face parte din soluţie;
- aria pătrăţelelor negre este mai mică decât *(aria pătratului)/k*; în acest caz îl divizăm în 4 pătrate disjuncte egale şi aplicăm acelaşi procedeu pe care l-am aplicat pentru pătratul iniţial

În cel mai defavorabil caz obţinem pătrate de latură egală cu 2 care conţin un singur pătrăţel negru. Acestea îndeplinesc condiţiile din enunţ (deoarece $1/k<1/4$).

Pentru a determina eficient numărul de pătrăţele negre putem folosi mai multe tehnici de *orthogonal range count*. Putem folosi range tree, pentru a răspunde la o întrebare în timp $O(log^2(N))$ (sau $O(log(N))$, dacă folosim *fractional cascading*). Ţinând cont de limitele coordonatelor punctelor, putem calcula o matrice P de dimensiuni LxL, unde $P(i,j)$ este egal cu numărul de puncte din dreptunghiul $(1,1)$-(i,j). Avem $P(i,j)=P(i,j-1)+P(i-1,j)-P(i-1,j-1)$ + (1, dacă pătrăţelul (i,j) este negru; 0, dacă este alb). Folosind această matrice, numărul de puncte negre dintr-un dreptunghi (a,b)-(c,d) (cu $a\leq b$ şi $c\leq d$) este egal cu $P(c,d)-P(c,b-1)-P(a-1,d)+P(a-1,b-1)$. Matricea se calculează în timp $O(L^2)$, iar răspunsul la o întrebare se dă în timp $O(1)$.

Problema 2-17. Tablete (Algoritmiada, infoarena 2008)

Construiţi o matrice A cu N linii şi N coloane care să conţină fiecare din numerele de la 1 la N^2 exact o dată, astfel încât:

- pe fiecare linie a matricii, numerele sunt în ordine crescătoare.
- pe coloana K ($2\leq K\leq N-1$), numerele trebuie să fie pare.

Soluţie: Vom începe prin a amplasa valorile de pe coloana K. Iniţializăm o variabilă $p=N\cdot K$ şi $row=N$. Cât timp $row\geq 1$ efectuăm următorii paşi:

1) dacă p este pară, atunci: {

1.1) $A(row,K)=p$;

1.2) vmin=p;

1.3) row=row-1};

2) $p=p-1$.

Vom completa apoi primele $K-1$ coloane. Iniţializăm o variabilă $next=0$. Parcurgem apoi liniile matricii A, în ordine, de la 1 la N: pentru fiecare rând *row*, parcugem coloanele i de la 1 la $K-1$.

Să presupunem că am ajuns la linia *row* şi coloana i. Mai întâi incrementăm variabila *next* cu 1. Apoi, cât timp *(next≥vmin)* şi *(next este pară)*, incrementăm variabila *next* cu 1. Vom seta apoi $A(row,i)=next$.

În continuare, vom completa coloanele $K+1,...,N$ ale matricii A. Iniţializăm variabila $next=N\cdot K$ (toate numerele de la 1 la $N\cdot K$ au fost amplasate pe primele K coloane ale matricii A). Parcurgem apoi matricea A pe linii şi, pentru fiecare linie *row*, parcurgem matricea pe coloane, de la $i=K+1$ la N.

Pentru o linie *row* şi o coloană i, incrementăm variabila *next* cu 1 şi apoi setăm $A(row,next)=1$.

La final, putem avea o problemă dacă $N\cdot K$ este un număr impar: în acest caz, vom rezolva problema interschimbând elementele de pe poziţiile *(linie=1, coloană=K+1)* şi *(linie=N,coloană=K)* din matricea A.

Problema 2-18. Împărţirea unui multiset în număr minim de submulţimi

Se dă un multiset S ce conţine N ($1 \le N \le 100.000$) numere: $S(1), ..., S(N)$. Împărţiţi aceste numere în cât mai puţine submulţimi, astfel încât elementele din fiecare submulţime să fie distincte două câte două. În caz că există mai multe împărţiri posibile cu număr minim de submulţimi, alegeţi-o pe aceea pentru care diferenţa dintre numărul maxim de elemente dintr-o submulţime şi numărul minim de elemente să fie cât mai mic.

Soluţie: Vom sorta numerele, astfel încât să avem $x(1)<x(2)<...<x(M)$ ($M \le N$) şi $cnt(i)$=numărul de apariţii ale elementului $x(i)$ în S. Putem realiza această sortare uşor, în timp $O(N \cdot log(N))$, sau, dacă numerele nu au valori foarte mari, putem folosi sortarea prin numărare (count sort).

Numărul minim de submulţimi necesare este $NS=max\{cnt(i)|1 \le i \le M\}$. În continuare, vom construi cele NS submulţimi: $SM(0), ..., SM(NS-1)$, după cum urmează. Le iniţializem pe toate la mulţimea vidă, apoi iniţializăm un contor $k=0$. Vom parcurge cele M elemente distincte şi, pentru fiecare element cu indicele i, vom realiza de $cnt(i)$ ori următorii paşi:

(1) adaugă $x(i)$ la submulţimea $SM(k)$;

(2) $k=(k+1) \ mod \ NS$.

În felul acesta, diferenţa dintre numărul maxim şi minim de elemente dintr-o submulţime este cel mult 1 (sau 0 dacă N este multiplu al lui NS). Datorită distribuirii de tip *round-robin* a elementelor în submulţimi, nu se va ajunge ca aceeaşi submulţime să conţină două elemente egale.

Problema 2-19. Secvenţă redusă (Olimpiada Baltică de Informatică, 2007)

Se dă o secvenţă $a(1), ..., a(N)$ ($1 \le N \le 1.000.000$). Asupra acestei secvenţe se poate efectua următoarea operaţie: $reduce(i)$=înlocuieşte elementele $a(i)$ şi $a(i+1)$ din secvenţă cu elementul $max\{a(i),a(i+1)\}$ ($1 \le i \le N-1$). În urma acestei operaţii, secvenţa va rămâne cu $N-1$ elemente. Costul operaţiei este $max\{a(i),a(i+1)\}$.

Determinaţi o secvenţă de $N-1$ operaţii *reduce* de cost minim (în urma cărora secvenţa iniţială este redusă la un singur element).

Soluţie: O soluţie bazată pe programare dinamică este evidentă. Fie $cmin(i,j)$=costul minim pentru a reduce subsecvenţa $a(i), a(i+1), ..., a(j)$ 1 aun singur element. Avem $cmin(i,i)=0$. Pentru $i<j$ (în ordine crescătoare a lui $j-i+1$) vom avea:

$cmin(i,j)=min\{cmin(i,k)+cmin(k+1,j)|i \le k \le j-1\}+Max(i,j)$.

$Max(i,j)$ este elementul maxim dintre $a(i), a(i+1), ..., a(j)$. Acest algoritm are complexitatea $O(N^3)$ şi este foarte ineficient pentru limitele date.

Un algoritm greedy este următorul. Fie $M=N-1$. Repetă de M ori următorii paşi:

1) găseşte orice indice i astfel încât $a(i-1) \ge a(i) \le a(i+1)$ (considerăm $a(0)=a(N+1)=+\infty$);

2) dacă $a(i-1)<a(i+1)$ atunci aplică operaţia $reduce(i-1)$; altfel aplică operaţia $reduce(i)$;

3) renumerotează elementele de la 1 la $N-1$ şi descreşte pe N cu 1.

O implementare directă a acestui algoritm are complexitate $O(N^2)$. O implementare cu complexitate $O(N)$ este următoarea. Vom parcurge secvenţa şi vom menţine o stivă S şi un cost C. Iniţializăm stiva astfel încât $S(1)=a(0)=+\infty$, $S(top=2)=a(1)$ şi $C=0$. Apoi vom considera poziţiile $i=2, 3, ..., N+1$.

Dacă $a(i)<S(top)$ atunci îl adăugăm pe $a(i)$ în vârful stivei:

1) $top=top+1$;

2) S(top)=a(i).

Altfel (*a(i)>S(top)*), atâta timp cât *condiţia de oprire* nu este îndeplinită, vom efectua în mod repetat următorii paşi:

1) dacă *S(top-1)<a(i)* atunci reduce elemente *S(top-1)* şi *S(top)* (şi creşte *C* cu *S(top-1)*), altfel reduce elementele *S(top)* şi *a(i)* (şi creşte *C* cu *a(i)*);

2) elimină *S(top)* din vârful stivei: *top=top-1*.

Dacă *i=N+1*, atunci condiţia de oprire este ca stiva să conţină doar *2* elemente (*top=2*), deoarece unul din ele este în mod sigur *a(0)* => secvenţa iniţială a fost redusă la un singur element. Dacă *i<N+1* atunci condiţia de oprire este următoarea: *a(i)<S(top)*.

La sfârşitul ciclului de reduceri, dacă *i<N+1*, atunci îl adăugăm pe *a(i)* în vărful stivei:

1) top=top+1;

2) S(top)=a(i).

O altă soluţie liniară foarte simplă este de a calcula *C* ca fiind egal cu suma valorilor *max{a(i),a(i+1)}* (*1≤i≤N-1*). Demonstraţia acestui fapt se bazează pe inducţie. Să presupunem că este adevărat pentru orice secvenţă cu *q≤k* elemente şi dorim acum să o demonstrăm pentru o secvenţă cu *k* elemente. Pentru *k=1* faptul este adevărat (costul fiind *0*).

Pentru *k≥2*, fie *p* poziţia unde se află elementul maxim *M* din secvenţă. Dacă *p=1* (*p=k*) atunci reducem secvenţa din dreapta (stânga) maximului, iar la final mai efectuăm o reducere de cost *M*. Dacă *2≤p≤k-1* vom reduce separat secvenţele din stănga şi din dreapta maximului, iar la sfârşit vom reduce de două ori maximum împreună cu cele *2* elemente rămase în stânga şi, respectiv, dreapta acestuia. În toate cazurile, costul obţinut de această strategie este egal cu valoarea calculată după regula menţionată mai sus.

Capitolul 3. Grafuri.

Problema 3-1. Sincrograf (Summer Trainings 2003 - Rusia, SGU)

Se dă un graf orientat cu *N* (*1≤N≤10.000*) noduri şi *M* (*0≤M≤100.000*) muchii, în care fiecare muchie *q* (*1≤q≤M*), orientată de la *i* la *j*, are ataşată o valoare *w(q)*.

Un nod *i* se numeşte *activ* dacă toate muchiile *q* care intră în *i* au *w(q)>0* (dacă nu există nicio astfel de muchie, nodul se consideră inactiv).

Din mulţimea nodurilor active, se poate selecta un nod *i* pentru a fi *declanşat*. Când un nod *i* este declanşat, se scade cu *1* valoarea *w(qin)* ataşată tuturor muchiilor *qin* care intră în nodul *i* şi se creşte cu *1* valoarea *w(qout)* a tuturor muchiilor *qout* care ies din nodul respectiv.

Un nod se numeşte *potenţial viu* dacă, pornind din starea curentă a grafului, există o secvenţă de declanşări, în urma căreia nodul *i* poate fi declanşat.

Un nod se numeşte *viu*, dacă este potenţial viu pornind din orice stare a grafului în care se poate ajunge începând din starea iniţială.

Determinaţi toate nodurile *vii* ale grafului.

Exemplu:

N=6, M=8 **muchia 1: (1->2), w(1)=1** **muchia 2: (4->3), w(2)=0** **muchia 3: (2->4), w(3)=0** **muchia 4: (4->3), w(4)=1** **muchia 5: (1->6), w(5)=0** **muchia 6: (6->3), w(6)=1** **muchia 7: (3->2), w(7)=0** **muchia 8: (4->5), w(8)=1000000000**	**Nodurile *vii* sunt nodurile 1 şi 6.**

Soluţie: Vom determina componentele tare conexe ale grafului dat (în complexitate *O(N+M)*). Dacă o componentă tare conexă SCC_i conţine un ciclu orientat C_{zero} în care toate muchiile au valoare *0*, atunci niciun nod din SCC_i nu este viu. Întâi, este evident că nodurile de pe acest ciclu nu pot fi vii, deoarece ele nu vor putea fi declanşate niciodată. Oricare din celelalte noduri face parte dintr-un ciclu orientat *C* care conţine un nod *x* care face parte şi din C_{zero}. Dacă declanşăm nodurile de pe *C* în ordine, ori de câte ori putem, începând cu succesorul *y* al nodului *x* pe ciclul *C*, vom ajunge într-o stare în care nu mai putem declanşa niciun nod de pe acest ciclu (deoarece muchia *(x,y)* va avea valoarea ataşată 0 şi nu va mai creşte niciodată).

Considerăm acum graful orientat aciclic al componentelor tare conexe. Vom sorta topologic aceste componente tare conexe, în ordinea SCC_1, ..., SCC_k. Vom reţine, pentru fiecare componentă, dacă este marcată ca fiind *moartă* sau nu. Dacă o componentă conţine un ciclu orientat cu muchii având valoare *0*, o vom marca ca fiind *moartă*.

Pentru a determina dacă o componentă tare conexă conţine un ciclu cu muchii de valoare *0*, procedăm în felul următor. Vom considera graful ce constă din nodurile componentei şi muchiile de valoare *0*. Pentru a determina dacă există un ciclu în acest graf, vom efectua parcurgeri DFS, după cum urmează. Iniţial marcăm toate nodurile ca fiind *nevizitate*. Apoi considerăm pe rând fiecare nod şi, dacă acesta este nevizitat, pornim o parcurgere DFS din el. Toate nodurile vizitate din cadrul parcurgerii DFS sunt marcate ca fiind *vizitate* imediat ce se

„intră" în ele. În cadrul parcurgerii, dintr-un nod *x* se poate ajunge direct doar în acele noduri *y* nevizitate încă către care există muchie orientată din *x*.

De asemenea, în cadrul parcurgerii, vom marca ca fiind *în stivă* toate nodurile aflate curent în stiva DFS (când se intră într-un nod, acesta este marcat ca fiind în stivă, iar când se iese dintr-un nod, acesta este demarcat, el nemaifiind în stivă). Dacă în cadrul parcurgerii, atunci când vizităm un nod *x*, găsim o muchie orientată de la un nod *x* la un nod *y* aflat *în stivă*, atunci am detectat un ciclu.

Complexitatea detectării ciclurilor de muchii cu valoare *0* pentru toate componentele tare conexe este *O(N+M)*.

Parcurgem apoi şirul componentelor tare conexe în ordinea *1,...,k*. Dacă SCC_i este marcată ca fiind moartă, vom marca ca fiind moarte toate componentele SCC_j ($j>i$) pentru care există o muchie oreintată (SCC_i->SCC_j) în graful orientat aciclic al componentelor tare conexe.

Toate nodurile din componente tare conexe care nu au fost marcate ca fiind *moarte* vor fi *vii*.

Problema 3-2. Secvenţă grafică

Se dă un şir format din *N* elemente naturale: *d(1), ..., d(N)* ($0 \le d(i) \le N-1$). Decideţi dacă există un graf neorientat cu *N* noduri astfel încât fiecare nod *i* ($1 \le i \le N$) să aibă gradul *d(i)*. Construiţi un astfel de graf.

Soluţie: O soluţie uşor de implementat este următoarea. Sortăm şirul de valori date astfel încât $d(1) \ge d(2) \ge ... \ge d(N)$ şi reţinem în vectorul *v* nodurile corespunzătoare (*v(i)* corespunde valorii *d(i)*).

Legăm nodul *v(1)* de nodurile *v(2), ..., v(2+d(1)-1)*. După aceea scădem cu *1* valorile lui *d(2), ..., d(2+d(1)-1)*, sortăm valorile de la *2* la *N* (modificând, în acelaşi timp, şi vectorul *v*, astfel încât, în orice moment, *v(i)* corespunde valorii *d(i)*). Efectuăm apoi acelaşi procedeu, considerând doar valorile *d(2), ..., d(N)*.

Practic, algoritmul va consta din *N* paşi. La fiecare pas *i* avem doar valorile *d(i), ..., d(N)*. Legăm nodul *v(i)* de nodurile *v(i+1), ..., v(i+d(i))*, scădem cu *1* valorile lui *d(i+1), ..., d(i+d(i))* şi resortăm valorile *d(i+1), ..., d(N)*.

Dacă, la un moment dat, avem $d(i)>N-i$ sau ajungem cu $d(i)<0$, atunci nu există soluţie. Acest algoritm se poate implementa uşor în complexitate $O(N^2 \cdot log(N))$ (*N* sortări). Totuşi, complexitatea se poate reduce la $O(N^2)$, deoarece, pentru a sorta valorile *d(i+1), ..., d(N)* la sfârşitul pasului *i*, este suficient să interclasăm două şiruri sortate: primul şir conţine valorile *d(i+1), ..., d(i+d(i))*, iar al doilea conţine valorile *d(i+d(i)+1), ..., d(N)*. Interclasarea se poate realiza în timp *O(N)*, obţinând, astfel, complexitatea precizată.

O altă metodă pentru a sorta valorile în timp *O(N)* la sfârşitul fiecărui pas este să ne folosim de faptul că valorile sunt numere întregi din intervalul *[0,N-1]*. Astfel, putem construi câte o listă *L(x)* (iniţial vidă) pentru fiecare valoare *x* ($0 \le x \le N-1$) şi inserăm valorile de sortat în lista corespunzătoare valorii. La final, doar concatenăm în ordinea *0, ..., N-1* cele *N* liste.

Evident, metoda descrisă mai sus poate fi folosită atât pentru a construi graful, cât şi pentru a decide dacă există un graf ale cărui noduri să aibă gradele date. Totuşi, există un algoritm liniar pentru problema de decizie. Sortăm în timp *O(N)* valorile *d(1), ..., d(N)*, astfel încât $d(1) \ge ... \ge d(N)$. Sortarea liniară se poate realiza folosind sortarea prin numărare. Avem acum următoarele două condiţii:

1) $\sum_{i=1}^{N} d(i)$ este pară.

2) $\sum_{i=1}^{k} d(i) < k \cdot (k-1) + \sum_{i=k+1}^{N} \min\{d(i),k\}$, pentru orice $k=1,...,N$.

Condiţia *(1)* este uşor de verificat în timp $O(N)$ (calculând sume prefix).

Pentru condiţia *(2)*, un algoritm cu timpul $O(N^2)$ este evident. Pentru a obţine un timp liniar, vom începe prin a calcula sumele prefix $sd(i)$: $sd(0)=0$ şi $sd(1 \le i \le n)=sd(i-1)+d(i)$. Vom porni apoi cu k de la N către 1, menţinând un vector cnt, unde $cnt[i]$ reprezintă numărul de valori $min\{d(j),k\}=i$ ($j>k$).

De asemenea, vom calcula $S(k)$, ca fiind suma din partea dreaptă a inegalităţii *(2)* (suma după i de la $k+1$ la N din $min\{d(i),k\}$). Evident, vom compara $sd(k)$ cu $k \cdot (k-1)+S(k)$, pentru fiecare valoare a lui k.

Pentru $k=N$ avem $S(N)=0$ şi $cnt[i]=0$ ($0 \le i \le N-1$). De fiecare dată când trecem de la $k=x$ la $k=x-1$, obţinem $S(x-1)$ ca fiind egală cu $S(x)-cnt[x]$. Apoi setăm $cnt[x-1]$ la $cnt[x-1]+cnt[x]$, după care resetăm $cnt[x]$ la 0. În felul acesta, am obţinut un algoritm cu complexitatea $O(N)$.

Probleme care au la bază secvenţe grafice au fost propuse la multe concursuri şi olimpiade de informatică (de ex., problema *Tennis* de la Olimpiada Baltică de Informatică, 2002).

Problema 3-3. Augmentarea unui graf orientat la un graf eulerian

Se dă un graf orientat având N ($1 \le N \le 50.000$) noduri şi M ($0 \le M \le 500.000$) muchii. Graful poate conţine mai multe muchii de la acelaşi nod către acelaşi alt nod, având acelaşi sens (muchii paralele). Adăugaţi un număr minim de muchii orientate acestui graf, astfel încât graful obţinut să conţină un ciclu (drum) eulerian.

Soluţie: Vom calcula, pentru fiecare nod i ($1 \le i \le N$), valorile $degin(i)$ şi $degout(i)$, reprezentând gradul de intrare (numărul de muchii care intră în nodul i), respectiv gradul de ieşire (numărul de muchii care ies din nodul i).

Apoi vom asocia fiecărui nod i valoarea $def(i)=degout(i)-degin(i)$.

Apoi vom ignora sensul muchiilor grafului şi vom împărţi graful în componente conexe: fie Q numărul de componente conexe ale sale. Dintre cele Q componente conexe ale sale, fie F numărul de componente conexe pentru care toate nodurile i din componentă au $def(i)=0$. Fiecare din celelalte Q-F componente conţine cel puţin un nod i cu $def(i)>0$ şi un nod j cu $def(j)<0$.

Vom ordona componentele într-o ordine oarecare şi le vom numerota de la 1 la Q. Din fiecare componentă i vom alege un nod $in(i)$ cu $def(i)>0$ şi un nod $out(i)$ cu $def(i)<0$ (dacă toate nodurile j din componentă au $def(j)=0$ atunci vom alege un nod k oarecare din componentă şi vom seta $in(i)=out(i)=k$).

În continuare vom adăuga muchiile orientate ($out(i)$->$in(i+1)$) ($1 \le i \le Q-1$), precum şi muchia orientată ($out(Q)$->$in(1)$). După adăugarea acestor muchii (şi actualizarea valorilor $degin(j)$, $degout(j)$ şi $def(j)$ ale nodurilor adiacente cu muchiile adăugate; mai exact, pentru fiecare muchie (a->b) adăugată incrementăm $degout(a)$, $degin(b)$ şi $def(a)$ cu 1, respectiv decrementăm $def(b)$ cu 1), vom considera S_1 mulţimea nodurilor i cu $def(i)<0$ şi S_2 mulţimea nodurilor i cu $def(i)>0$.

Vom nota prin $S_j(k)$ al k-lea nod din mulţimea S_j ($k \ge 1$) şi prin $|S_j|$ numărul de elemente din S_j.

Vom iniţializa două contoare $p_1=p_2=1$. Cât timp $p_1 \le |S_1|$ şi $p_2 \le |S_2|$:

1) adăugăm muchia $(S_1(p_1)$->$S_2(p_2))$;

2) incrementăm *degout*($S_1(p_1)$), *degin*($S_2(p_2)$) şi *def*($S_1(p_1)$) cu *1*;
3) vom decrementa *def*($S_2(p_2)$) cu *1*;
4) dacă *def*($S_1(p_1)$)*=0* atunci $p_1=p_1+1$;
5) dacă *def*($S_2(p_2)$)*=0* atunci $p_2=p_2+1$.

Numărul de muchii adăugate este egal cu *V+F*, unde *V* este suma valorilor iniţiale *def(i)*, pentru acele noduri *i* cu *def(i)>0*.

Problema 3-4. Gangsteri (Olimpiada Baltică de Informatică, 2003, enunţ modificat)

Într-un oraş există *N* ($1 \le N \le 10.000$) gangsteri, organizaţi în grupe de prieteni. Se cunosc *M* ($0 \le M \le 500.000$) relaţii între *M* perechi de gangsteri, de forma:

1) gangsterii *i* şi *j* sunt prieteni sau
2) gangsterii *i* şi *j* sunt duşmani ($1 \le j \le N$).

Se ştie că gangsterii au următorul cod de etică:

1) Prietenul prietenului meu îmi este prieten.
2) Duşmanul duşmanului meu îmi este prieten.

Pe baza celor *M* relaţii date, determinaţi care este numărul maxim de grupe de prieteni ce pot exista.

Exemplu:

N=6, M=4 **1 şi 4 sunt duşmani.** **3 şi 5 sunt prieteni.** **4 şi 6 sunt prieteni.** **1 şi 2 sunt duşmani.**	**Răspuns: 3** **Cele 3 grupe de prieteni sunt: {1}, {2,4,6}, {3,5}.**

Soluţie: Vom construi un graf *GF* în care fiecare nod corespunde unui gangster. Vom introduce în acest graf muchie între doi gangsteri *i* şi *j* dacă se ştie că aceştia sunt prieteni (ori sunt prieteni direct, ori au un duşman comun). Problema cere determinarea numărului de componente conexe din graful *GF*.

Introducerea muchiilor *(i,j)* pentru relaţiile de prietenie este simplă. Pentru relaţiile de duşmănie, vom construi un al doilea graf, *GE*, în care introducem muchie între două noduri *i* şi *j* dacă ştim că cei doi sunt duşmani. Apoi, pentru fiecare nod *i*, vom construi o listă cu toţi vecinii lui *i* în *GE*, *lv(i)*. Oricare două noduri din *lv(i)* reprezintă doi gangsteri care sunt prieteni (deoarece au un duşman comun), astfel că am putea introduce o muchie *(a,b)* între oricare două noduri *a* şi *b* din *lv(i)*. Dacă implementăm algoritmul în felul acesta, putem ajunge la o complexitate prea mare. Să observăm că obţinem acelaşi efect şi în felul următor. Alegem un nod *a* din *lv(i)* şi adăugăm muchie în *GF* între *a* şi orice nod $b \neq a$ din *lv(i)*.

În cazul în care ar fi necesară şi verificarea corectitudinii datelor, va trebui doar să verificăm că oricare două noduri *i* şi *j* între care există muchie în GE nu sunt în aceeaşi componentă conexă în *GF*.

Problema 3-5. Împărţirea pătrăţelelor

Se dă o matrice cu *NxN* pătrăţele. Fiecare pătrăţel este colorat în alb (*0*) sau în negru (*1*). Fie *Q* numărul de pătrăţele negre. Determinaţi dacă este posibil să împărţim pătrăţelele negre în două mulţimi disjuncte, egale, *A* şi *B*, astfel încât mulţimea *A* să poate fi obţinută din mulţimea *B* prin translaţii şi rotaţii cu multipli de 90^o.

Soluţie: Vom considera toate posibilităţile *(tl, tc)* de translaţii pe linii şi coloane ($-N \le tl,tc \le N$). Pentru o pereche *(tl,tc)* fixată, vom încerca toate cele 4 posibilităţi de rotaţii *r* (cu 0^o, 90^o, 180^o şi 270^o).

Pentru fiecare tuplu *(tl, tc, r)*, vom parcurge toate cele *Q* pătrăţele negre şi vom determina „succesorul" fiecăruia. Succesorul unui pătrăţel de pe linia *(i,j)* se calculează în felul următor: rotim pătrăţelul *(i,j)* conform rotaţiei *r* şi obţinem un pătrăţel *(i',j')*. De exemplu, pentru rotaţia cu 90^o: $i'=N-j+1$ şi $j'=i$. Apoi translatăm coordonatele *(i',j')* cu *(tl,tc)*, obţinând coordonatele $(i''=i'+tl, j''=j'+tc)$. Dacă pătrăţelul *(i'',j'')* se află în matrice (adică $1 \le i'', j'' \le N$) şi este de culoare neagră, setăm succesorul lui *(i,j)* la *(i'',j'')*; altfel, *(i,j)* nu va avea succesor.

Astfel, am obţinut un graf orientat, în care fiecare nod are gradul de ieşire *0* sau *1* (muchia orientată de la nod la succesorul acestuia). Acest graf este compus din cicluri sau din drumuri. Dacă numărul de noduri de pe fiecare ciclu este par şi numărul de noduri de pe fiecare drum este par, atunci am găsit o soluţie la problema noastră. În acest caz, mulţimea *A* se obţine luând nodurile din *2* în *2* de pe fiecare drum sau ciclu, iar mulţimea B se obţine din nodurile celelalte. Este clar că mulţimea *B* se poate roti şi translata peste mulţimea *A*.

Aşadar, problema admite soluţie dacă găsim un tuplu *(tl, tc, r)* pentru care graful construit are numai cicluri şi drumuri cu număr par de noduri. Complexitatea algoritmului este $O(N^4)$.

Problema 3-6. Split Graphs

Un graf neorientat cu *N* ($1 \le N \le 1.000$) noduri şi *M* ($0 \le M \le N \cdot (N-1)/2$) muchii se numeşte *split graph*, dacă nodurile sale pot fi împărţite în *2* submulţimi *C* şi *I* (eventual vide), astfel încât:

(1) pentru oricare două noduri *x* şi *y* din *C* există o muchie *(x,y)* în graf;

(2) pentru oricare două noduri *x* şi *y* din *I*, muchia *(x,y)* nu există în graf.

Mai exact, nodurile din *C* formează o clică (subgtaf complet), iar nodurile din *I* formează o mulţime independentă (sau intern stabilă). Determinaţi (dacă există) o împărţire a nodurilor în cele două mulţimi *C* şi *I* (evident, fiecare nod trebuie să facă parte din una din cele două mulţimi).

Soluţie: Vom asociat fiecărui nod *i* ($1 \le i \le N$) o variabilă *x(i)*, ce va indica mulţimea din care face parte nodul (dacă $x(i)=1$, atunci nodul *i* face parte din mulţimea *C*; dacă $x(i)=0$, atunci nodul *i* face parte din mulţimea *I*).

Să considerăm un nod *s* oarecare din graf. Vom considera două cazuri. În primul caz, vom alege $x(s)=1$, iar în al doilea caz vom alege $x(s)=0$. În fiecare caz vom introduce nodul *s* într-o coadă *Q*. Valorile $x(i \neq s)$ vor avea o valoare specială, reprezentând faptul că nu au fost iniţializate.

Apoi, cât timp *Q* nu este vidă, vom extrage nodul *a* din vârful cozii. Dacă $x(a)=1$, atunci vom considera toate nodurile *b* pentru care nu există muchia *(a,b)* în graf: dacă *x(b)* este *neiniţializat*, setăm $x(b)=0$ (nodul *b* nu poate face parte din mulţimea *C*) şi introducem nodul *b* în coadă; dacă *x(b)* este iniţializat şi este diferit de *0*, atunci nu vom avea soluţie în acest caz (valoarea iniţială *x(s)* nu a fost „ghicită" corect), iar dacă $x(b)=0$, atunci nu întreprindem nicio acţiune. Dacă $x(a)=0$, atunci vom considera toate nodurile *b* pentru care muchia *(a,b)* există în graf: dacă *x(b)* este *neiniţializat*, atunci setăm $x(b)=1$ (nodul *b* nu poate face parte din mulţimea *I*) şi introducem nodul *b* în *Q*; dacă *x(b)* este iniţializat şi diferit de *1*, atunci nu avem soluţie pentru acest caz, iar dacă $x(b)=1$, atunci mergem mai departe.

La finalul execuţiei acestui pas, dacă nu s-a găsit nicio contradicţie, avem următoarea situaţie. Unele noduri *i* au variabila *x(i)* setată la *1* sau *0*, iar altele o pot avea încă neiniţializată. Nodurile *j* cu *x(j)* încă neiniţializat sunt adiacente cu toate nodurile *i* cu *x(i)=1* şi nu sunt adiacente cu niciun nod *k* cu *x(k)=0*. Aşadar, valorile variabilelor lor (*x(j)*) pot fi alese independent de valorile variabilelor deja setate. Prin urmare, vom considera un graf redus ce constă doar din nodurile cu variabiele *x(i)* neiniţializate. Din acest graf vom alege un nod oarecare *s* şi vom considera, ca şi prima dată, două cazuri (*x(s)=0* şi *x(s)=1*), propagând restricţiile mai departe. Vom relua execuţia algoritmului descris atâta timp cât mai există variabile *x(j)* neiniţializate.

Dacă reuşim să ajungem la final fără nicio contradicţie, atunci valorile x(i) indică apartenenţa nodurilor la cele două mulţimi. Complexitatea algoritmului este $O(N^2)$. Fiecare nod este introdus o singură dată în coadă şi pentru fiecare nod extras din coadă se consideră *O(N)* alte noduri.

Problema 3-7. Bază de cicluri a unui graf

Se dă un graf neorientat (nu neapărat conex) având *N* noduri ($1 \le N \le 30.000$) şi *M* muchii ($0 \le M \le min\{200.000, N \cdot (N-1)/2\}$). Determinaţi o mulţime *Q* conţinând un număr maxim de cicluri simple din acest graf, astfel încât fiecare ciclu să conţină cel puţin o muchie care nu apare în niciun alt ciclu din *Q*. Ne interesează doar numărul de elemente din *Q*, nu şi ciclurile propriu-zise.

Soluţie: Pentru fiecare componentă conexă vom determina un arbore DFS al acesteia. Să presupunem că componenta *i* (*1≤i≤ncc*; *ncc*=numărul de componente conexe ale grafului) conţine *nn(i)* noduri şi *mm(i)* muchii. În arborele DFS se găsesc *nn(i)-1* muchii. Dacă adăugăm fiecare din celelalte *(mm(i)-nn(i)+1)* muchii la arborele DFS al componentei conexe *i*, acestea vor închide câte un ciclu în arbore (format din muchia adăugată *(a,b)* şi drumul unic dintre nodurile *a* şi *b* din arbore). Mulţimea *Q* este formată din ciclurile închise de fiecare muchie *(a,b)* în arborele DFS al componentei sale (muchia *(a,b)* se află în afara arborelui DFS al componentei sale).

Se observă uşor că fiecare astfel de ciclu corespunzător unei muchii *(a,b)* are drept muchie unică (care nu apare în alte cicluri) chiar pe muchia *(a,b)*. Astfel, numărul total de cicluri din *Q* este suma valorilor *(mm(i)-nn(i)+1)* (*1≤i≤ncc*). Cum suma valorilor *mm(i)* este *M* şi suma valorilor *nn(i)* este *N* (*1≤i≤ncc*), rezultatul este: *M-N+ncc*.

Aşadar, trebuie doar să determinăm numărul de componente conexe din graful dat. Putem realiza acest lucru în modul standard. Reţinem întreg graful în memorie, apoi efectuăm parcurgeri DFS sau BFS din câte un nod nevizitat, marcând ca vizitate toate nodurile parcurse. Totuşi, dacă numărul de muchii este prea mare pentru ca graful să fie menţinut complet în memorie, putem proceda după cum urmează. Vom menţine o structură de mulţimi disjuncte. Iniţial, fiecare nod *x* este o mulţime separată. Pe măsură ce citim câte o muchie *(a,b)*, unim mulţimile în care se află nodurile *a* şi *b*. Numărul de mulţimi obţinute la final este chiar numărul de componente conexe. Folosind o implementare arborescentă a mulţimilor disjuncte [CLRS], complexitatea acestei soluţii este $O(M \cdot log^*(N))$. Vom iniţializa numărul de componente conexe *ncc* la *N*. La fiecare muchie *(a,b)* determinăm întâi dacă *a* şi *b* sunt în aceeaşi mulţime sau nu. Dacă nu sunt în aceeaşi mulţime, doar atunci efectuăm unirea celor două mulţimi şi decrementăm *ncc* cu *1*.

Problema 3-8. Muchii Esenţiale (Olimpiada de Informatică a Europei Centrale, 2005)

Se dă un graf neorientat conex având *N* noduri ($1 \leq N \leq 30.000$) şi *M* muchii ($0 \leq M \leq min\{200.000, N \cdot (N-1)/2\}$). Unele noduri oferă servicii de tipul *A*, iar altele de tipul *B* (pentru fiecare nod se ştie ce fel de servicii oferă; este posibil ca un nod să ofere ambele tipuri de servicii).

O muchie se numeşte *esenţială* dacă prin eliminarea ei din graf există cel puţin un nod care nu mai are acces (cale directă) către niciun nod care oferă un serviciu de tipul *A* sau de tipul *B*.

Determinaţi muchiile *esenţiale* ale grafului.

Soluţie: Vom aplica întâi algoritmul clasic de determinare a muchiilor critice, bazat pe o parcurgere DFS a grafului. O muchie esenţială trebuie neapărat să fie şi o muchie critică. Muchiile critice sunt muchii ce fac parte din arborele DFS obţinut în urma parcurgerii. Mai rămâne să determinăm care dintre muchiile critice sunt şi esenţiale.

Vom calcula pentru fiecare nod *i* din arborele DFS numerele $nA(i)$ şi $nB(i)$, reprezentând numărul de noduri ce oferă servicii de tipul *A*, respectiv *B*, din subarborele nodului *i* (aceste valori se calculează ca sume ale valorilor corespunzătoare din fiii lui *i*, la care, eventual, se adaugă *1*, dacă nodul *i* oferă servicii de tipul respectiv).

Fie *NTA* şi *NTB* numărul total de noduri ce oferă servicii de tipul *A*, respectiv *B*. O muchie critică (x,y) (cu *x* părintele lui *y* în arborele DFS) este esenţială dacă:

(1) $nA(x)=NTA$; sau (2) $nA(x)=0$; sau (3) $nB(x)=NTB$; sau (4) $nB(x)=0$.

Complexitatea algoritmului este $O(N+M)$.

Problema 3-9. Depozit (Olimpiada de Informatică a Europei Centrale, 2005)

Într-un depozit se află aşezate în linie $N \cdot M$ produse. Produsele sunt de tipurile *1, 2, ..., M* şi există *N* produse din fiecare tip. Dorim să ordonăm produsele, astfel încât în orice interval de poziţii $[(i-1) \cdot M+1,\ i \cdot M]$ ($1 \leq i \leq N$) să existe câte un produs din fiecare tip (altfel spus, pe primele *M* poziţii să se afle *M* produse distincte între ele, pe următoarele *M* poziţii să se afle tot *M* produse distincte între ele ş.a.m.d.).

Depozitul are şi o poziţie liberă, localizată (iniţial) la sfârşitul celor $N \cdot M$ produse (deci, pe poziţia $N \cdot M+1$). Pentru ordonarea produselor putem efectua doar următorul tip de mutări: luăm un produs de pe o poziţie *i* şi îl mutăm pe poziţia liberă; evident, după efectuarea mutării, poziţia *i* devine noua poziţie liberă. La final, poziţia liberă trebuie să fie, ca şi la început, poziţia $N \cdot M+1$. În plus, numărul total de mutări efectuate trebuie să fie minim.

Soluţie: Vom construi un graf bipartit orientat ce conţine vârfurile $p(1), ..., p(N)$ în partea stângă şi vărfurile $q(1), ..., q(N)$ în partea dreaptă. Între 2 vârfuri $p(i)$ şi $q(j)$ există *k* muchii orientate de la $p(i)$ la $q(j)$, dacă produsul *j* apare de $k+1$ ori în intervalul de poziţii $[(i-1) \cdot M+1,\ i \cdot M]$. De asemenea, există muchie orientată de la $q(j)$ la $p(i)$ dacă produsul *j* nu apare niciodată în intervalul $[(i-1) \cdot M+1,\ i \cdot M]$.

Pentru fiecare nod *x* al grafului, numărul de muchii care intră în *x* este egal cu numărul de muchii care ies din *x*. Astfel, fiecare componentă (conexă) a grafului conţine un ciclu eulerian. Să presupunem că am determinat un astfel de ciclu pentru o componentă. Alegem prima muchie ca fiind de tipul $(p(a),q(b))$ şi mutăm unul din produsele de tipul *b* din intervalul $[(a-1) \cdot M+1,\ a \cdot M]$ în poziţia liberă $N \cdot M+1$. Vom parcurge ciclul în sens invers (începând de la muchia dinaintea primei muchii). Pentru fiecare două muchii consecutive de pe ciclu (în sens invers) $(q(j), p(i))$ şi $(p(k), q(j))$, vom muta unul din produsele de tipul *j* din

intervalul $[(k-1)\cdot M+1, k\cdot M]$ pe poziţia liberă (care se află în intervalul $[(i-1)\cdot M+1, i\cdot M]$); după aceasta, noua poziţie liberă se va afla undeva în intervalul $[(k-1)\cdot M+1, k\cdot M]$.

La final, vom mai avea de tratat perechea de muchii consecutive $(p(a),q(b))$ (muchia iniţială) şi $(q(b), p(c))$. Vom muta produsul de pe poziţia $N\cdot M+1$ pe poziţia liberă, care se află undeva în intervalul $[(c-1)\cdot M+1, c\cdot M]$.

Numărul de mutări efectuate este minim şi este egal cu *ncc* + suma gradelor de ieşire ale nodurilor $p(*)$ (*ncc*=numărul de componente conexe ale grafului; o componentă conexă se calculează ignorând direcţia muchiilor). Complexitatea algoritmului este $O(N\cdot M)$.

Problema 3-10. Cereri (Olimpiada de Informatică a Europei Centrale 2008)

Se dau N ($1\leq N\leq 200$) cereri de procesare. Fiecare cerere i ($1\leq i\leq N$) aduce un venit $V(i)$ (dacă este acceptată), iar pentru procesarea ei are nevoie de alocarea calculatoarelor din mulţimea $C(i)$ ($C(i)$ este o submulţime a mulţimii $\{1,...,M\}$, unde $1\leq M\leq 200$ este numărul total de calculatoare disponibile). Pentru a folosi un calculator i pentru cererile care au nevoie de el, avem două opţiuni:

1) putem cumpăra calculatorul i, la preţul $P(i)$;

2) putem închiria calculatorul i pentru fiecare cerere j acceptată pentru care i face parte din $S(j)$, la preţul $R(j,i)$.

Cererile pot fi acceptate sau rejectate. Dacă sunt acceptate, atunci calculatoarele cerute trebuie alocate. Dacă un calculator i este cumpărat, atunci el poate fi folosit pentru toate cererile care au nevoie de el (nu trebuie închiriat niciodată); dacă un calculator i nu este închiriat, atunci trebuie plătit preţul $R(j,i)$ pentru fiecare cerere j acceptată care are nevoie de calculatorul i.

Determinaţi profitul maxim ce poate fi obţinut (*profit=venituri totale – costuri totale*), valorile tuturor veniturilor şi costurilor sunt distincte.

Soluţie: Vom construi următorul graf bipartit. În partea stângă se află câte un nod $x(i)$ ce corespunde fiecărei cereri i; în partea dreaptă se află câte un nod $y(j)$ ce corespunde fiecărui calculator j. Avem o muchie orientată de la $x(i)$ la $y(j)$, dacă j face parte din $S(i)$. Mai adăugăm o sursă *Sr*, care are muchii orientate de la ea către fiecare nod $x(i)$ şi o destinaţie virtuală *De*, pentru care există muchii de la fiecare nod $y(j)$ la *De*. Fiecare muchie a grafului are o anumită capacitate. Fiecare muchie $(Sr, x(i))$ are capacitatea $V(i)$; fiecare muchie $(x(i), y(j))$ are capacitatea $R(i,j)$; fiecare muchie $(y(j), De)$ are capacitatea $P(j)$.

Vom determina un flux maxim în această reţea de flux. Vom nota prin $F(a,b)$ valoarea fluxului pe o muchie (a,b) a grafului. Dacă o muchie $(Sr,x(i))$ are $F(Sr,x(i))=V(i)$, atunci vom rejecta cererea i; altfel, o vom accepta. Dacă o muchie $(y(j),De)$ are $F(y(j),De)=P(j)$, atunci vom cumpăra calculatorul j. Pentru fiecare cerere i acceptată şi fiecare calculator j din $S(i)$, dacă j nu a fost cumpărat, atunci el va fi închiriat pentru cererea i, plătind preţul $R(i,j)$. În felul acesta, veniturile şi cheltuielile au fost determinate şi profitul poate fi calculat imediat.

Problema 3-11. Paintball (Lotul Naţional de Informatică, România 2008)

Se dă un graf cu N ($1\leq N\leq 100.000$) noduri. Din fiecare nod u iese exact o muchie, care este îndreptată către un alt nod v (deci fiecare nod al grafului are gradul de ieşire 1). Să considerăm o permutare P a nodurilor grafului: $P(1), ..., P(N)$. Considerând ordinea din permutare, fiecare nod u trage către nodul v către care este îndreptată muchia de ieşire din u şi îl omoară. Un nod u trage atunci când îi vine rândul doar dacă nu a fost omorât de alt nod în prealabil şi dacă nodul v către care vrea să tragă nu este deja mort.

În mod evident, numărul de noduri moarte variază în funcţie de permutarea aleasă. Determinaţi care este numărul maxim şi numărul minim de morţi (dacă alegem corespunzător permutarea nodurilor).

Soluţie: Pentru a determina numărul minim de morţi vom proceda după cum urmează. Vom menţine un contor *ndead* (iniţial *0*) şi o coadă în care introducem iniţial doar nodurilr care au gradul interior *0*. Pe rând, vom extrage din coadă un nod *u* şi vom determina nodul *v* către care este îndreptată muchia care iese din *u*. Dacă *v* nu e mort, atunci *u* trage către *v* şi îl omoară: incrementăm *ndead* cu *1* şi eliminăm din graf atât nodul *u*, cât şi nodul *v* => asta înseamnă că decrementăm cu *1* gradul de intrare al nodului *w* către care era îndreptată muchia care ieşea din *v*; dacă gradul de intrare al lui *w* devine *0* (şi *w* nu e deja mort), atunci îl introducem pe *w* în coadă. După această primă etapă, în graf au mai rămas, eventual, doar cicluri. Pentru fiecare ciclu de lungime *L* putem alege orice nod să tragă primul. După aceasta, ciclul s-a transformat într-un lanţ cu *L-1* noduri, din care vor rezulta *(L-1) div 2* morţi. Astfel, pentru un ciclu de lungime *L* numărul minim de morţi este *((L+1) div 2)*; incrementăm *ndead* cu această valoare pentru fiecare ciclu. La final, *ndead* va conţine numărul minim de morţi.

Pentru a determina numărul maxim de morţi vom iniţializa contorul *ndead* cu *0* şi apoi vom considera, pe rând, fiecare nod *u* cu grad de intrare *0*. Vom determina toate nodurile care pot fi atinse din nodul *u*, mergând în sensul muchiilor (facem o parcurgere BFS sau DFS din *u*, fără a trece prin noduri vizitate în cadrul altor pacurgeri anterioare). Aceste noduri formează un lanţ, în care ultimul nod are o muchie îndreptată către un nod din lanţ situat înaintea sa (sau către un nod vizitat într-o parcurgere anterioară). Dacă lanţul conţine *L* noduri, atunci putem obţine *L-1* morţi (trage întâi penultimul nod, apoi antepenultimul, ş.a.m.d.; singurul care nu omoară pe nimeni este ultimul nod din lanţ şi singurul care nu moare este primul nod din lanţ); incrementăm *ndead* cu *L-1*.

După acest pas, în graf au mai rămas, eventual, cicluri. Pentru fiecare ciclu de lungime *L*, numărul maxim de morţi este *L-1*. Alegem orice nod să tragă la început, după care rămânem cu un lanţ de lungime *L-1*, din care, după cum am văzut, putem obţine *L-2* morţi; incrementăm *ndead* cu *L-1* pentru fiecare ciclu (a cărui lungime este *L*). Numărul maxim de morţi este conţinut în contorul ndead.

Problema 3-12. Orientarea muchiilor unui graf planar (CPSPC 2007)

Se dă un graf planar cu *N* ($1 \leq N \leq 10.000$) noduri şi *M* muchii. Dorim să atribuim o orientare fiecărei muchii *(u,v)* a grafului (*u->v* sau *v->u*), astfel încât din fiecare nod să iasă cel mult *3* muchii.

Soluţie: Vom începe cu o orientare oarecare a muchiilor grafului. Apoi vom parcurge nodurile grafului. De fiecare dată când găsim un nod *i* care are mai mult de *3* muchii care ies din el, efectuăm următoarele acţiuni: cât timp nodul *i* are mai mult de *3* muchii de ieşire, efectuăm o parcurgere *BFS* din nodul *i*. Pentru aceasta, vom folosi o coadă în care, iniţial, introducem nodul *i*. Parcurgem nodurile din coadă şi, dacă nodul *u* din vârful cozii are cel puţin *3* muchii de ieşire, introducem în coadă toţi vecinii săi *v* către care există muchia *u->v* (dacă nu au fost deja vizitaţi). Dacă are mai puţin de *3* muchii de ieşire, îl introducem într-o mulţime specială *Q* (şi nu introducem în coadă vecinii săi *v* către care existau muchii orientate de la *u*).

Din parcurgerea efectuată am obţinut un arbore BFS. Eliminăm, pe rând, din acest arbore, acele frunze care au *3* sau mai multe muchii de ieşire în graf. Astfel, rămânem cu un arbore în care toate frunzele au *0*, *1* sau *2* muchii de ieşire în graf. Vom inversa sensul tuturor muchiilor din arbore (de la *tată->fiu*, inversăm sensul la *fiu->tată*). Numărul muchiilor de ieşire în graf ale frunzelor creşte cu *1*, iar numărul muchiilor de ieşire ale nodurilor interne scade sau rămâne constant. Numărul muchiilor de ieşire din rădăcină (nodul *i* de la care am început parcurgerea) scade.

Vom repeta parcurgerea dacă nodul *i* are, în continuare, mai mult de *3* muchii de ieşire. Algoritmul va găsi de fiecare dată o soluţie, deoarece un graf planar are cel mult $3 \cdot N-6$ muchii şi, deci, permite existenţa unei orientări a muchiilor cu proprietatea specificată. Mai exact, algoritmul ar putea să nu funcţioneze doar dacă, la un moment dat, se elimină toate nodurile din arborele BFS (adică toate nodurile din graf care pot fi vizitate din nodul *i* au cel puţin *3* muchii de ieşire). Întrucât o muchie de ieşire pentru un nod este o muchie de intrare pentru un alt nod, ar trebui ca între cele *M* noduri ce pot fi atinse din nodul *i* (inclusiv nodul *i*) să existe cel puţin $3 \cdot M$ muchii (plus *1*, căci nodul *i* are cel puţin *4* muchii de ieşire), număr care depăşeşte limita maximă de $3 \cdot M-6$.

Complexitatea algoritmului este $O(N^2)$ în cel mai rău caz, însă, în practică, merge mult mai bine.

Problema 3-13. Link (Olimpiada de Informatică a Europei Centrale, 2006)

Se dă un graf orientat cu *N* ($2 \leq N \leq 500.000$) noduri. Din fiecare nod *u* iese exact o singură muchie, care îl uneşte de alt nod *v* (este posibil ca *u=v*); evident, muchia este orientată de la *u* la *v* (o vom nota prin *(u,v)*). Nodul *1* este un nod special. Dorim să adăugăm un număr minim de muchii grafului, astfel încât lungimea minimă a unui drum orientat de la nodul *1* către orice alt nod să fie cel mult *K* ($1 \leq K \leq 20.000$). Un drum orientat de la *u* la *v* porneşte din *u* şi continuă pe una din muchiile care ies din *u*, ş.a.m.d., până ajunge în nodul *v*. Lungimea unui drum este egală nu numărul de muchii de pe drum.

Soluţie: În prima fază a algoritmului vom marca ca fiind *acoperite* cele (maxim) *K* noduri care sunt la distanţă cel mult *K* de nodul *1*, inclusiv nodul *1* (vom urma maxim *K* muchii de ieşire, pornind din nodul *1*; întrucât fiecare nod are exact o singură muchie de ieşire, acest pas are complexitatea *O(K)*). Toate celelalte noduri vor fi marcate ca fiind *neacoperite*. Să considerăm acum componentele „conexe” ale grafului, ignorând sensul muchiilor. Fiecare astfel de componentă conţine un ciclu în interiorul său şi nişte arbori ale căror rădăcini sunt nodurile de pe ciclu. Arborii respectivi au muchiile orientate de la fiu către părinte.

În prima etapă vom trata nodurile care sunt în interiorul arborilor „ataşaţi” nodurilor din ciclul fiecărei componente. Vom menţine o valoare *degin[u]* pentru fiecare nod *u*, reprezentând câte muchii *(v,u)* există în graf (practic, gradul de intrare al nodului *u*). Vom introduce într-o coadă *Q* toate nodurile *u* neacoperite, pentru care *degin[u]=0*.

Fiecărui nod *u* introdus în *Q* îi vom asocia o distanţă *dist[u]*, reprezentând distanţa de la nodul *1* la acest nod. Pentru nodurile *u* neacoperite care au iniţial *degin[u]=0* vom introduce muchiile *(1,u)* în graf şi vom seta *dist[u]=1*; apoi le vom marca ca fiind *acoperite*. Pentru toate nodurile *v* marcate ca fiind acoperite până acum, vom seta *dist[v]*=lungimea drumului orientat de la nodul *1* la nodul *v* (*dist[1]=0*).

Vom extrage apoi, pe rând, nodurile din *Q*, până când *Q* devine goală. Să presupunem că am extras un nod *u*. Fie *next(u)* nodul către care se îndreaptă muchia care iese din *u* (muchia orientată *(u,next(u))* există în graf). Dacă $dist[u]<K$ şi ((*next(u)* este *neacoperit*) sau

(*dist[next(u)]>dist[u]+1*)), atunci vom seta *dist[next(u)]=dist(u)+1* şi îl vom marca pe *next(u)* ca fiind *acoperit* (vom considera că, iniţial, pentru fiecare nod *v* neacoperit avem *dist[v]*=+∞). Apoi vom decrementa *degin[next(u)]*. Dacă *degin[next(u)]=0*, atunci: dacă *next(u)* este încă *neacoperit*, atunci vom introduce o muchie suplimentară orientată *(1,next(u))* şi vom seta *dist[next(u)]=1*; apoi vom introduce nodul *next(u)* în *Q*.

Observăm că în cadrul acestei etape au fost acoperite toate nodurile care nu aparţin ciclului din componenta proprie (precum şi, eventual, rădăcinile arborilor ataşati ciclului din fiecare componentă; aceste rădăcini sunt, după cum am spus anterior, noduri din cadrul ciclului).

Vom considera acum fiecare ciclu în parte din cadrul fiecărei componente. Fie acest ciclu format din nodurile *v(1), ..., v(P)* (*P*≥*1*). Unele din nodurile *v(i)* de pe ciclu pot fi *acoperite* şi au deja o valoare corectă pentru *dist[v(i)]*. Celelalte noduri *v(j)* nu sunt acoperite şi nu au setată valoarea *dist[v(j)]*. Dacă niciunul din nodurile de pe ciclu nu este acoperit, atunci vom alege nodul *v(1)* şi vom introduce muchia suplimentară *(1,v(1))* (după care setăm *dist[v(1)]=1* şi îl marcăm pe *v(1)*) ca fiind *acoperit*. Astfel, în continuare, vom presupune că ciclul conţine cel puţin un nod acoperit.

Vom selecta acel nod acoperit *u* de pe ciclu pentru care *dist[u]* este minim dintre toate celelalte noduri de pe ciclu. Vom numerota nodurile de pe ciclu astfel încât *v(1)=u*. Vom parcurge apoi nodurile de pe ciclu în sensul muchiilor de ieşire, de la *i=2,…,P* (avem *v(i)=next(v(i-1))*). Când ajungem la un nod *v(i)*, dacă *v(i-1)* este *acoperit* şi *dist[v(i-1)]<K*, atunci:

(1) dacă nodul *v(i)* este *neacoperit*, setăm *dist[v(i)]=dist[v(i-1)]+1* şi marcăm *v(i)* ca fiind acoperit;

(2) dacă nodul *v(i)* este *acoperit* (şi *v(i-1)* este şi el acoperit, cu *dist[v(i-1)]<K*), atunci setăm *dist[v(i)]=min{dist[v(i)], dist[v(i-1)]+1}*.

După această parcurgere este posibil să mai fi rămas noduri neacoperite pe ciclu, printre care se află unele noduri acoperite. Pentru acoperirea nodurilor neacoperite va trebui să introducem muchii suplimentare.

Pentru acoperirea nodurilor neacoperite, va trebui să alegem un nod iniţial *v(j)* neacoperit, de pe ciclu. Vom introduce muchia *(1,v(j))*, îl vom marca pe *v(j)* ca fiind acoperit, apoi vom avansa pe ciclu *K-1* poziţii şi vom marca toate nodurile întâlnite şi neacoperite încă ca fiind acoperite; să presupunem că am ajuns la poziţia *r*. Cât timp mai sunt noduri acoperite pe ciclu (putem menţine această informaţie sub forma unui contor de noduri neacoperite), vom avansa pe ciclu în continuare (începând de la poziţia *r*), până la următorul nod neacoperit; fie acest nod *v(l)*; vom introduce muchia suplimentară orientată *(1,v(l))*, vom seta *dist[v(l)]=1* şi îl vom marca pe *v(l)* ca fiind acoperit; apoi vom avansa *K-1* poziţii pe ciclu începând de la poziţia *l* şi vom marca toate nodurile întâlnite şi neacoperite încă ca fiind acoperite; fie *r* noua poziţie la care am ajuns pe ciclu (refolosim variabila *r*). Aşadar, o dată ce am ales nodul neacoperit iniţial *v(j)*, putem acoperi optim celelalte noduri în timp *O(P)*.

Problema care apare este că numărul de muchii suplimentare introduse depinde de nodul *v(j)* ales iniţial. Va trebui să alegem acel nod *v(j)* pentru care se introduce un număr minim de muchii suplimentare.

O primă variantă ar fi să încercăm orice nod neacoperit ca punct de start (am obţine o complexitate $O(P^2)$ pe ciclu, şi $O(N^2)$ în total). O îmbunătăţire constă în a încerca să plecăm doar din nodurile de start neacoperite dintr-un interval de *K* noduri de pe ciclu (adică găsim primul nod neacoperit *v(j)*, şi apoi considerăm doar nodurile *v(i)* cu *j*≤*i*≤*j*+*K*, unde adunarea

poziţia $j+K$ se calculează circular pe ciclu); această abordare are o complexitate de $O(P \cdot K)$ pe ciclu şi $O(N \cdot K)$ per total.

Putem îmbunătăţi algoritmul după cum urmează. Vom modifica ciclul în $u(0), ..., u(L-1)$, pentru a conţine doar nodurile neacoperite. Vom avea muchii orientate $(u(i), u((i+1) \bmod L))$, de lungime $len(u(i), u((i+1) \bmod L))=1$ + numărul de noduri acoperite aflate între nodurile $u(i)$ şi $u((i+1) \bmod L)$ pe ciclul iniţial. Vom considera pe rând fiecare nod $i=0,...,L-1$ şi vom menţine 2 variabile: D şi idx (iniţial $D=0$ şi $idx=0$). Pentru fiecare nod $u(i)$, dacă $i>0$, vom seta $D=D-len(u(i-1+L) \bmod L, u(i))$. Apoi, cât timp $D<K$ şi $idx \neq i$, vom seta:

(1) $D=D+len(u(idx), u((idx+1) \bmod L))$;

(2) $idx=(idx+1) \bmod L$.

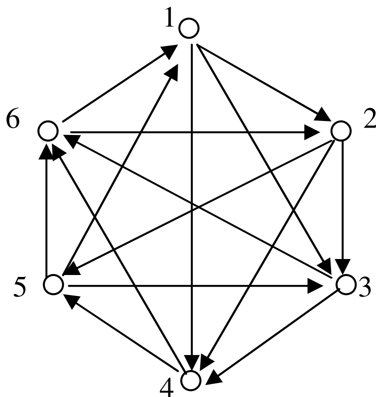
La sfârşitul ciclului vom seta $Jump[i]=idx$, având semnificaţia că $u(Jump[i])$ este primul nod neacoperit de pe ciclu (mergând în sensul ciclului începând de la nodul $u(i)$), ce nu poate fi acoperit dacă se introduce muchia suplimentară $(1,u(i))$. $Jump(u[i])$ sare peste K noduri de pe ciclul iniţial (sau mai puţin de K, dacă nu există atâtea noduri pe ciclul iniţial).

Astfel, atunci când pornim cu un nod de start neacoperit $u(j)$, vom sări direct în $u(Jump[j])$, fără a trece prin nodurile dintre cele 2 noduri de pe ciclu; vom continua să sărim din nodul la care am ajuns, $u(o)$, la nodul următor $u(Jump[o])$, până când ajungem înapoi în nodul iniţial $u(j)$ (sau până îl depăşim cu ultima săritură). Numărul de muchii suplimentare necesare, dacă plecăm din nodul $u(j)$, va fi egal cu 1 + *numărul de sărituri* – 1 (se consideră prima muchie suplimentară $(1,u(j))$ şi se exclude ultima săritură prin care ajungem înapoi la $u(j)$ sau sărim peste el). Complexitatea pentru un nod de start $u(j)$ este acum $O(P/K)$. Întrucât pentru fiecare ciclu considerăm $O(K)$ noduri de start, complexitatea totală este $O(P)$ pentru fiecare ciclu, şi $O(N)$ per total.

Problema 3-14. Oraş (infoarena)

Se dă un graf neorientat complet cu N noduri ($1 \leq N \leq 10.000$) noduri. Determinaţi o orientare pentru fiecare din muchiile sale, astfel încât între oricare 2 noduri din graf x şi y să existe un drum orientat de la x la y care să conţină cel mult 2 muchii.

Soluţie: Pentru $N=2$ şi $N=4$ nu există soluţii. Dacă N este impar, pornim de la graful cu nodurile 1, 2 şi 3, în care avem muchiile orientate astfel: $1\text{->}2$, $2\text{->}3$, $3\text{->}1$. Dacă N este par, vom porni de la un graf cu 6 noduri, care respectă proprietatea (un astfel de graf este desenat mai jos):



Să presupunem că avem un graf cu K noduri (numerotate de la 1 la K) care are proprietatea dorită (şi K are aceeaşi paritate ca N). Cât timp $K<N$, efectuăm următorii paşi. Adăugăm muchii orientate de la nodul $K+1$ la toate nodurile $1,...,K$ şi muchii orientate de la fiecare din nodurile $1,...,K$ la nodul $K+2$. Apoi adăugăm muchia orientată $(K+2)$->$(K+1)$. Observăm că avem acum un graf cu $K+2$ noduri care respectă proprietatea cerută; astfel, vom seta $K=K+2$. Complexitatea acestui algoritm este liniară în numărul de muchii ale grafului rezultat ($O(N^2)$).

Problema 3-15. Biomech (Happy Coding 2006, infoarena)

În anul 2006, oamenii au construit primul robot biomecanic cu inteligenţă artificială. În orice caz, încă nu se ştie prea bine cât de avansată este inteligenţa sa. Tocmai de aceea, robotul va fi supus unui test. El va fi plasat într-o zonă rectangulară, împarţită în pătrate amplasate pe *5* linii şi un număr infinit de coloane. Coloanele sunt numerotate de la $-\infty$ la $+\infty$ şi robotul este plasat iniţial în coloana cu numărul *0*. Liniile zonei rectangulare sunt numerotate de la 1 la 5 şi robotul va fi plasat la început în linia 3 (cea din mijloc). Robotul va fi orientat în una din cele 8 direcţii posibile: Nord, Nord-Est, Est, Sud-Est, Est, Sud, Sud-Vest, Vest, Nord-Vest.

Mutările pe care robotul le poate face sunt:

- Rotatie cu un unghi multiplu de *45* de grade

Din direcţia spre care este îndreptat, robotul se poate întoarce astfel spre oricare altă direcţie. O rotaţie de la o direcţie iniţială la o direcţie finală consumă o anumită cantitate de timp. Din cauza structurii interne a robotului, se poate ca o rotaţie cu un unghi mai mare să dureze mai puţin decât o rotaţie cu un unghi mic. De asemenea, o rotaţie din direcţia X în direcţia Y s-ar putea să nu dureze la fel de mult ca o rotaţie din direcţia Y în direcţia X.

- Mişcare în direcţia spre care este orientat, din pătratul curent în următorul patrat (având o muchie comună sau un vârf comun cu acesta)

De exemplu, dacă roboţelul este la linia *3*, coloana X, orientat spre Nord-Est, s-ar putea deplasa în pătrăţelul de pe linia *2*, coloana $X+1$. După mutare, robotul nu îşi schimbă direcţia în care este orientat. De asemenea, nu îi este permis să se mute în afara zonei rectangulare (aşadar, anumite mişcări sunt interzise din anumite pătrate).

Cantitatea de timp necesară pentru o mutare depinde atât de direcţia în care se mută (din cauza câmpului magnetic al Pământului), cât şi de linia pe care robotul se află în momentul curent (deoarece fiecare dintre cele *5* rânduri are o structura electromagnetică diferita). Totuşi, costurile mutărilor nu depind de coloana în care se află robotul.

Robotul va fi supus unui test, după cum urmează: i se vor acorda *TMAX* ($0 \leq TMAX \leq 10^{15}$) unităţi de timp. Folosindu-le, va trebui să mute cât mai departe posibil de poziţia curentă. Distanţa nu este măsurată în termeni de pătrate, ci în termeni de coloane. Dacă după *TMAX* unităţi temporale robotul se află pe coloana X, distanţa este considerată $|X|$ (valoarea absolută a lui X). Nu este importantă linia pe care ajunge.

Robotul va alege direcţia în care va fi orientat iniţial şi cantitatea de timp va începe să scadă după ce ia această decizie. Aflaţi care este distanţa maximă pe care robotul o poate parcurge în *TMAX* unităţi de timp.

Soluţie: Vom calcula matricea *A[lstart][dirstart][lfinish][dirfinish][P]*, reprezentând timpul minim pentru a ajunge de pe linia *lstart*, o coloană oarecare X şi privind în direcţia *dirstart*, până pe linia *lfinish*, coloana $X+2^P$ (deci o coloană situată cu 2^P poziţii mai în dreapta) şi privind în direcţia *dirfinish*. Pentru $P=0$, vom folosi un algoritm de drum minim. Va trebui să

avem grijă, căci soluţia de timp minim pentru a trece de pe o coloană pe coloana imediat din dreapta ei poate presupune nişte mutări „in spate”, pe un număr limitat de coloane din stânga. Alegând o limită maximă de *9* coloane în stânga şi în dreapta, putem folosi algoritmul lui Dijkstra pe un graf ale cărui noduri constau din elementele unei submatrici de *5* linii, *19* coloane şi au *8* orientări. Pentru $P>0$, avem:

A[lstart][dirstart][lfinish][dirfinish][P] = A[lstart][dirstart][lintermed][dirintermed][P-1] + A[lintermed] [dirintermed][lfinish][dirfinish][P-1].

Variem linia şi orientarea intermediară şi păstrăm minimul. În mod similar, vom calcula o matrice *B[lstart][dirstart][lfinish][dirfinish][P]*, unde indicele *P* va reprezenta sosirea pe o coloană situată cu 2^P coloane la stânga coloanei de start.

Cu aceste matrici calculate, vom determina numărul maxim de coloane pe care le poate parcurge robotul spre stânga şi spre dreapta, în timpul dat, alegând maximul dintre cele *2* variante. Voi prezenta în continuare doar cazul deplasării spre dreapta, cel al deplasării spre stânga fiind similar. Vom porni de la coloana *0* şi vom mări treptat numărul de coloane pe care le poate parcurge robotul (fie acest numar *C*). Pentru fiecare valoare a lui *C* şi fiecare stare posibilă *(linie,orientare)* vom menţine timpul minim în care se poate ajunge în starea respectivă. Iniţial, robotul poate ajunge în *0* unităţi de timp doar în starea iniţială. Vom porni cu *P* de la valoarea maximă pentru care am calculat valori (de exemplu, această valoare poate fi *61*) şi îl vom decrementa treptat. Presupunând că am ajuns până la coloana *C*, vom încerca acum să parcurgem încă 2^P coloane. Folosind matricea *A* şi cunoscând timpul minim pentru a ajunge la coloana *C* în fiecare stare posibilă S_1, vom calcula timpul minim pentru a ajunge la coloana $C+2^P$, pentru fiecare stare S_2. Dacă cel puţin o stare S_2 are acest moment de timp mai mic sau egal cu durata de timp maximă dată, atunci mărim valoarea lui *C* cu 2^P şi vom considera momentele de timp actualizate pentru noua valoare a lui *C*. Dacă pentru nicio stare nu se poate ajunge la coloana $C+2^P$ în timp util, atunci lăsăm valoarea lui *C* nemodificată, nefăcând nimic altceva (la următorul pas având, însă, o valoare a lui *P* cu *1* mai mică). Valoarea finală a lui *C* este numărul maxim de coloane ce poate fi parcurs spre dreapta in timpul dat.

Problema 3-16. Mutarea bilelor

Se dă un graf cu *N* ($1\leq N\leq 200$) noduri. Fiecare nod *i* ($1\leq i\leq N$) conţine $b(i)\geq 0$ bile. Putem să efectuăm mutări conform cărora luăm o bilă dintr-un nod *i* şi o mutăm într-un vecin *j* al acestuia; costul unei astfel de mutări este *c(i,j)* (*c(i,j)* poate fi diferit de *c(j,i)*). Determinaţi costul total minim al mutărilor ce trebuie efectuate pentru ca, la final, fiecare nod *i* să conţină exact *q(i)* bile (suma valorilor *b(i)* este egală cu suma valorilor *q(i)*).

Soluţie: Vom calcula distanţele minime (ca număr de muchii) dintre oricare *2* noduri din graf. Fie *d(i,j)* distanţa minimă dintre nodurile *i* şi *j*. Vom construi un graf bipartit, după cum urmează. În partea stângă vom plasa nodurile *i* care au $b(i)>q(i)$, iar în partea dreaptă nodurile *j* care au $b(j)<q(j)$. Vom duce muchii orientate de capacitate $+\infty$ şi cost *d(i,j)* de la fiecare nod *i* din partea stângă la fiecare nod *j* din partea dreaptă. Apoi vom adăuga o sursă virtuală *S*, împreună cu muchiile orientate de la *S* la fiecare nod *i* din partea stângă; fiecare astfel de muchie va avea cost *0* şi capacitate egală cu *b(i)-q(i)*. Vom adăuga şi o destinaţie virtuală *T*, împreună cu muchiile orientate de la fiecare nod *j* din partea dreaptă la nodul *T*; fiecare astfel de muchie va avea cost *0* şi capacitate *q(j)-b(j)*. Vom calcula un flux maxim de cost minim de la *S* la *T* în graful astfel construit. Costul minim al acestui flux va reprezenta

numărul minim de mutări ce trebuie efectuate pentru ca fiecare nod i să ajungă, la final, să conţină exact $q(i)$ bile.

Problema poate fi generalizată după cum urmează: la final, fiecare nod i trebuie să conţină cel puţin $low(i)$ şi cel mult $high(i)$ bile. Acest caz poate fi rezolvat în mod similar. Construim un graf bipartit similar, în care în partea stângă punem toate nodurile i care au $b(i)>low(i)$, iar în partea dreaptă toate nodurile j care au $b(j)<low(j)$. Ducem din nou muchii orientate de capacitate $+\infty$ şi cost $d(i,j)$ între fiecare nod i din partea stângă şi fiecare nod j din partea dreaptă. Apoi introducem sursa virtuală S şi destinaţia virtuală T. Ducem muchii de la S la fiecare nod i din partea stângă; fiecare astfel de muchie va avea cost 0, capacitate superioară egală cu $b(i)-low(i)$ şi o capacitate inferioară egală cu $max\{0, b(i)-high(i)\}$. Ducem şi muchii orientate de la fiecare nod j din partea dreaptă la destinaţia virtuală T; fiecare astfel de muchie va avea cost 0 şi capacitate superioară egală cu $high(j)-b(j)$ şi capacitatea inferioară egală cu $low(j)-b(j)$. Vom găsi un flux fezabil (care respectă constrângerile de capacităţi inferioare şi superioare) de la S la T în acest graf, al cărui cost să fie minim. Pentru aceasta, vom efectua transformarea standard pentru a calcula un flux fezabil cu capacităţi inferioare şi superioare, care are la bază modificarea grafului şi reducerea la o problemă de flux maxim [CLRS]; păstrând aceleaşi costuri pe muchii, în această etapă vom calcula un flux maxim de cost minim (în loc de un flux maxim simplu).

Problema 3-17. Augmentarea unui arbore

Se dă un arbore cu N ($1\le N\le 100.000$) noduri. Adăugaţi un număr minim de muchii la arbore, astfel încât, la finalm fiecare nod să facă parte din exact un ciclu. Considerăm două cazuri:

(a) se pot „dubla" muchiile arborelui (adică se pot adăuga muchii identice cu muchiile arborelui) şi

(b) nu se pot „dubla" muchiile arborelui.

Soluţie: Vom alege o rădăcină r a arborelui, definind astfel relaţii tată-fiu. Apoi vom calcula, pentru fiecare nod i, următoarele valori:

- $CA(i)$=numărul minim de muchii necesare pentru ca întreg subarborele nodului i să fie satisfăcut (adică fiecare nod din subarborele nodului face parte din exact un ciclu, fără a considera noduri din afara subarborelui)
- $CB(i)$=numărul minim de muchii necesare pentru ca întreg subarborele nodului i să fie satisfăcut, mai puţin nodul i
- $CC(i)$=numărul minim de muchii necesare pentru ca întreg subarborele nodului i să fie satisfăcut, mai puţin un lanţ care începe la nodul i şi coboară în jos în subarborele acestuia până la un nod diferit de i (nu neapărat până la o frunză)

Vom calcula aceste valori pornind de la frunze spre rădăcină. Pentru o frunză i avem următoarele: $CA(i)=+\infty$, $CB(i)=0$, $CC(i)=+\infty$. Pentru un nod i care nu este frunză, fie fiii acestuia: $f(i,1), ..., f(i,nfii(i))$. Avem:

- $CB(i)$=suma valorilor $CA(f(i,j))$ ($1\le j\le nfii(i)$)
- $CC(i)=CB(i)+min\{min\{CB(f(i,j), CC(f(i,j))\} - CA(f(i,j))\} \mid 1\le j\le nfii(i)\}$

Pentru a calcula $CA(i)$ vom proceda după cum urmează. Calculăm mai întâi valoarea $CA'(i)=1+CC(i)$ (pentru cazul *(a)*) sau $CA'(i)=1+CB(i)+min\{CC(f(i,j)) - CA(f(i,j)) \mid 1\le j\le nfii(i)\}$ (pentru cazul *(b)*).

Apoi, dacă $nfii(i)\ge 2$, vom asocia fiecărui fiu $f(i,j)$ o valoare $CF(f(i,j))=min\{CB(f(i,j), CC(f(i,j))\}-CA(f(i,j))$. Vom selecta (în timp $O(nfii(i))$) cei doi fii $f(i,a)$ şi $f(i,b)$ ($a\ne b$) pentru

care valorile asociate $CF(f(i,a))$ şi $CF(f(i,b))$ sunt cele mai mici două valori dintre valorile tuturor fiilor. Calculăm apoi $CA''(i)=1+CB(i)+CF(f(i,a))+CF(f(i,b))$ (sau, dacă $nfii(i)=1$, $CA''(i)=+\infty$).

Vom avea $CA(i)=min\{CA'(i), CA''(i)\}$.

Răspunsul este $CA(r)$. Complexitatea întregului algoritm este $O(N)$.

Capitolul 4. Structuri de Date.

Problema 4-1. Arbore Cartezian (ACM ICPC NEERC, Northern, 2002)

Se dau *N* ($1 \le N \le 100.000$) puncte în plan; punctul *i* ($1 \le i \le N$) are coordonatele (x_i,y_i). Un *arbore cartezian* al celor *N* puncte are proprietatea de arbore binar de căutare pentru coordonata *x* şi de min-heap pentru coordonata *y*. Mai exact, fie *Q* un nod al arborelui cartezian, ce corespunde unui punct (x_Q,y_Q):

- x_Q este mai mare sau egal decât coordonatele *x* ale tuturor punctelor din subarborele stâng şi mai mic sau egal decât coordonatele *x* ale tuturor punctelor din subarborele drept
- y_Q este mai mică decât coordonatele *y* ale punctelor din subarborele stâng sau drept

Construiţi un arbore cartezian pentru punctele date.

Soluţie: Vom sorta punctele după coordonata lor *x*, astfel incât $x_1 \le \ldots \le x_N$. Construcţia unui arbore cartezian se poate realiza recursiv, apelând *CartTree* cu parameterii *1* şi *N*.

<u>CartTree(i,j):</u>
if $(i=j)$ **then return** $((x_i,y_i),(),())$
else if $(i>j)$ **then return** ()
else {
  selectează poziţia *p*, astfel încât y_p este valoarea minimă dintre valorile y_k ($i \le k \le j$)
  $T_{stânga}$=**CartTree**$(i,p-1)$
  $T_{dreapta}$=**CartTree**$(p+1,j)$
  return $((x_p,y_p),T_{stânga},T_{dreapta})$

Pentru fiecare din cele $O(n)$ noduri ale arborelui cartezian se execută operaţia de determinare a valorii minime dintr-un interval de indici. Dacă această valoare este căutată în timp liniar, complexitatea algoritmului este $O(n^2)$. Dacă folosim, însă, RMQ, şi calculăm în timp $O(1)$ indicele valorii minime dintr-un interval dat, atunci complexitatea finală devine $O(N \cdot log(N))$ (de la preprocesarea necesară pentru RMQ şi sortarea iniţială a celor *N* puncte).

Problema 4-2. Coada (Olimpiada de Informatică a Europei Centrale, 2006)

Avem o coadă cu *X* ($1 \le X \le 10^9$) elemente. Fiecare element are o etichetă, egală cu poziţia sa (iniţială) în coadă (aşadar, elementele sunt numerotate de la *1* la *X*, începând de la capătul din stânga al cozii către cel din dreapta).

Se efectuează *N* ($0 \le N \le 50.000$) operaţii de felul următor: *OP(A,B)*=se scoate din coadă elementul cu eticheta *A* şi se inserează în coadă înaintea (în stânga) elementului cu eticheta *B*.

După efectuarea celor *N* operaţii, se dau *Q* ($1 \le Q \le 50.000$) întrebări dintr-unul din următoarele *2* tipuri:

1) pe ce poziţie în coadă se află elementul etichetat cu *A* ?

2) ce etichetă are elementul aflat pe poziţia *B* în coadă ?

Soluţie: Pentru fiecare element *p* din coadă definim *pred(p)* şi *succ(p)*, predecesorul şi succesorul lui *p* în coadă (pentru elementul *p'* de pe prima poziţie definim *pred(p')=0* şi pentru elementul *p''* de pe ultima poziţie definim *succ(p'')=X+1*; vom menţine şi valorile *succ(0)* şi *pred(X+1)*, definite ca *1* şi, respectiv *X*, iniţial).

Putem împărţi cele X elemente în două submulţimi U şi V. Elementele p din mulţimea U au proprietatea că $pred(p)=p-1$ şi $succ(p)=p+1$ (adică se află chiar în starea iniţială, ele nefiind propriu-zis afectate de mutările realizate). Elementele din mulţimea V sunt acele noduri afectate de mutări. Vom menţine elementele din mulţimea V sub forma unui arbore echilibrat, iniţial vid (în acest arbore vom ţine şi valorile $pred(p)$ şi $succ(p)$ ale unui element p).

De fiecare dată când efectuăm o mutare (luăm elementul A şi îl punem în faţa elementului B), efectuăm următoarele acţiuni:

1) $succ(pred(A))=succ(A)$;
2) $pred(succ(A))=pred(A)$;
3) $succ(pred(B))=A$;
4) $pred(B)=A$;
5) $pred(A)=pred(B)$;
6) $succ(A)=B$.

Când vrem să calculăm $pred(p)$ sau $succ(p)$, îl căutăm pe p în mulţimea V. Dacă îl găsim acolo, atunci vom lua de acolo valorile $succ(p)$ şi $pred(p)$; dacă nu îl găsim, vom considera că $succ(p)=p+1$ şi $pred(p)=p-1$. Atunci când facem o atribuire $succ(p)=z$ sau $pred(p)=z$, întâi îl căutăm pe p în mulţimea V. Dacă îl găsim, efectuăm atribuirea. Dacă nu îl găsim, îl inserăm pe p în z cu $pred(p)=p-1$ şi $succ(p)=p+1$. După aceasta îl căutăm din nou pe p în V şi efectuăm atribuirea. Aşadar, partea de procesare a mutărilor se poate realiza în timp $O(N \cdot log(N))$.

Pentru a răspunde la întrebări, vom sorta întrebările separat, în funcţie de tip. Întrebările de tipul *1* vor fi sortate crescător după eticheta elementului interogat, iar cele de tipul *2* crescător după poziţia interogată. Apoi vom parcurge coada comprimată, menţinând, în timpul parcurgerii, numărul nel de elemente peste care am trecut (iniţial, $nel=0$) şi ultimul element p din mulţimea V peste care am trecut (iniţial, $p=0$).

Cât timp $p<X$, în mod repetat vom proceda după cum urmează. Fie $q=succ(p)$. Dacă q este în mulţimea V, atunci incrementăm nel cu *1*. Căutăm apoi binar în mulţimea întrebărilor de tipul *1*, dacă există vreo întrebare pentru elementul cu eticheta q; dacă da, atunci răspunsul la întrebările de tipul *1* cu eticheta q va fi nel. În mod similar, căutăm binar între întrebările de tipul *2* dacă există vreo întrebare pentru poziţia nel; dacă da, răspunsul la toate întrebările de tipul *2* pentru poziţia nel este q. Apoi setăm $p=q$.

Dacă, însă, q nu se află în mulţimea V, atunci vom căuta în mulţimea V cel mai mic element q' strict mai mare decât q (dacă nu există un astfel de element, setăm $q'=X+1$). Ştim acum că intervalul de elemente $[q,q'-1]$ se află în aceeaşi ordine ca şi cea iniţială. Vom căuta în mulţimea întrebărilor de tipul *1* prima întrebare cu o etichetă $e \geq q$. Cât timp $e \leq q'-1$ punem răspunsul la întrebare ca fiind $nel+1+(e-q)$, după care trecem la următoarea întrebare din şirul sortat (şi vom seta e la eticheta noii întrebări). Vom căuta apoi în mulţimea întrebărilor de tipul *2* prima întrebare cu o poziţie $poz>nel$. Cât timp $poz \leq nel+(q'-q)$, punem răspunsul la întrebare ca fiind $poz-nel+q-1$, după care trecem la următoarea întrebare (şi setăm poz la poziţia noii întrebări). După ce răspundem la toate aceste întrebări, incrementăm nel cu $(q'-q+1)$, poziţionându-ne astfel pe elementul q'. Dacă $q' \leq X$, atunci îl vom trata ca mai sus (răspunzând la întrebările cu eticheta q' sau poziţia nel). La finalul iteraţiei setăm $p=q'$.

Observăm că a doua parte (de răspuns la întrebări) poate fi rezolvată in timp $O(Q \cdot log(Q)+N \cdot (log(N)+log(Q)))$.

Problema 4-3. Planificarea activităţilor

Aveţi la dispoziţie un procesor şi N ($1 \leq N \leq 100.000$) activităţi. Fiecare activitate necesită o unitate de timp de procesare, timp în care foloseşte procesorul exclusiv (nu poate fi executată o altă activitate simultan). Fiecare activitate i are asociat un profit $p(i)$ ($1 \leq p(i) \leq 100.000$) şi un timp limită $tf(i)$ ($1 \leq tf(i) \leq 100.000$) până la care trebuie încheiată execuţia activităţii respective. Determinaţi o submulţime S de activităţi care să fie executate pe procesor, astfel încât fiecare activitate i din S să îşi încheie execuţia până la momentul $tf(i)$, iar suma profiturilor activităţilor din S să fie maximă.

Soluţie: Vom prezenta o primă soluţie, care are complexitatea $O(N \cdot log(N))$. Vom sorta momentele de timp $tf(i)$ în ordine crescătoare şi vom elimina duplicatele, rămânând cu $M \leq N$ momente de timp $t(1) < ... < t(M)$. Fiecărui moment $t(j)$ îi vom asocia o listă $L(j)$ în care vom introduce toate activităţile i cu $tf(i)=t(j)$ (atunci când sortăm momentele de timp reţinem şi indicele activităţii respective, iar o listă este formată din activităţile i consecutive în ordinea sortată a momentelor de timp care au aceeaşi valoare $tf(i)$).

În cadrul fiecărei liste $L(j)$ vom sorta activităţile descrescător după profit. Să presupunem că lista $L(j)$ are $na(j)$ activităţi: $L(j,1), ..., L(j,na(j))$, astfel încât $p(L(j,1)) \geq ... \geq p(L(j,na(j)))$. Dacă $na(j) > t(j)$, vom trunchia $na(j)$ la $t(j)$.

Vom parcurge momentele de timp de la cel mai mic la cel mai mare şi vom menţine un min-heap cu activităţile planificate temporar spre a fi executate pe procesor. Cheia asociată fiecărei activităţi este profitul său. Iniţial, heap-ul este gol.

Când ajungem la un moment $t(j)$, ştim că până la acest moment pot fi planificate cel mult $t(j)$ activităţi. Vom menţine un indice k al activităţii curente din lista $L(j)$. Iniţial, $k=1$. Cât timp $k \leq na(j)$ şi heap-ul nu conţine $t(j)$ activităţi, vom adăuga activitatea $L(j,k)$ în heap (cu cheia $p(L(j,k))$) şi vom incrementa pe k cu 1. La sfârşitul acestei faze, ori avem $k=na(j)+1$, ori heap-ul conţine exact $t(j)$ activităţi. Dacă $k \leq na(j)$, executăm următorii paşi: cât timp $k \leq na(j)$ şi cel mai mic profit $pmin$ al unei activităţi din heap este strict mai mic decât $p(L(j,k))$, eliminăm din heap activitatea cu profitul cel mai mic şi introducem în heap activitatea $L(j,k)$ (cu cheia $p(L(j,k))$). În felul acesta, activităţile planificate anterior, dar care au profituri mici, sunt înlocuite cu alte activităţi mai profitabile şi care pot fi încă planificate pe procesor în timpul alocat.

La final, activităţle i din heap vor fi planificate pe procesor, în ordine crescătoare a momentelor de timp $tf(i)$. Complexitatea fiecărei operaţii de adăugare în şi de ştergere din heap este $O(log(N))$, iar complexitatea totală este $O(N \cdot log(N))$.

O altă soluţie tot de complexitate $O(N \cdot log(N))$ este următoarea. Vom sorta activităţile în ordine descrescătoare a profitului. Dacă două activităţi au profituri egale, o vom plasa pe cea cu timpul limită mai mic înaintea celei cu timpul limită mai mare. Vom parcurge activităţile în această ordine. Vom încerca să planificăm fiecare activitate i pe procesor astfel încât să se termine la cel mai mare moment de timp $t \leq tf(i)$ care este disponibil. Pentru aceasta, va trebui să menţinem o structură de date care să permită găsirea acestui moment de timp.

Vom considera momentele de timp de la 1 până la $Tmax=max\{tf(j) | 1 \leq j \leq N\}$. Iniţial, toate aceste momente de timp sunt disponibile. O primă variantă constă în construirea unui arbore de intervale peste aceste $Tmax$ momente de timp. Fiecare moment de timp corespunde unei frunze din arborele de intervale. Fiecare nod intern al arborelui de intervale va conţine numărul de momente de timp disponibile dintre frunzele din subarborele său. Atunci când dorim să găsim cel mai mare moment de timp disponibil dintr-un interval $[a,b]$, vom

determina acele noduri interne ale arborelui a căror reuniune a intervalelor (disjuncte) acoperă complet intervalul *[a,b]*. Vom considera aceste noduri din arbore de la cel cu intervalul cel mai din dreapta, către cel cu intervalul cel mai din stânga. Pentru fiecare nod *q*, verificăm dacă conţine cel puţin un moment de timp disponibil. Dacă da, atunci mergem recursiv în subarborele acestuia, uitându-ne la fiii lui *q*. Dacă fiul drept conţine cel puţin un moment disponibil, mergem mai departe (recursiv) în fiul drept; altfel mergem (recursiv) în fiul stâng. Dacă nu găsim niciun moment de timp disponibil, activitatea respectivă nu poate fi planificată.

După găsirea unui moment de timp *t*, vom parcurge toate nodurile din arbore de pe drumul de la frunza corespunzătoare lui *t* până la rădăcina arborelui şi vom decrementa numărul de momente de timp disponibile din aceste noduri. Operaţiile de găsire a unui moment de timp disponibil şi de „ocupare" aunui moment de timp au complexitatea *O(log(N))* fiecare.

În locul unui arbore de intervale putem folosi o împărţire în grupuri (disjuncte) de câte (aproximativ) *sqrt(N)* momente de timp consecutive fiecare. Fiecare grup menţine numărul de momente de timp disponibile din interiorul său şi pentru fiecare moment de timp ştim dacă e disponibil sau nu. Când căutăm cel mai din dreapta moment de timp disponibil dintr-un interval *[a,b]*, vom determina grupele intersectate de acest interval. Fie grupa din stânga *GL*, grupa din dreapta *GR* şi grupele interioare *GI(j)* (*1≤j≤ngrint*). Vom parcurge momentele de timp de la *b* până la începutul lui *GR* (în ordine descrescătoare) şi vom căuta primul moment disponibil. Dacă nu găsim niciun astfel de moment, parcurgem grupele interioare *GI(j)* de la dreapta la stânga, căutând o grupă care să conţină cel puţin un moment disponibil. Dacă găsim o astfel de grupă, parcurgem momentele de timp din interiorul ei de la dreapta la stânga. Dacă tot nu am găsit un moment disponibil, parcurgem momentele de timp din *GL*, de la cel mai din dreapta moment din *GL*, până la momentul de timp *a*. Complexitatea operaţiei de căutare este *O(sqrt(N))*. Pentru a marca un moment de timp *t* ca fiind ocupat, îl marcăm ca ocupat în vectorul caracteristic folosit (*ocupat[t]=true*) şi decrementăm cu *1* numărul de momente disponibile din grupa *G* ce conţine momentul de timp *t* (pentru fiecare moment *t* se menţine şi o asociere *grupa[t]=G*, creată în momentul construcţiei grupelor).

O a treia variantă (după arborele de intervale şi împărţirea în grupuri de câte *sqrt(N)* momente de timp), este următoarea. Vom menţine intervalele de momente de timp ocupate sub forma unor mulţimi disjuncte. Când un moment de timp *t* este marcat ca fiind ocupat, se creează o mulţime ce conţine momentul *t*. Apoi se verifică dacă momentul *t-1* (*t≥2*) este ocupat. Dacă da, se caută reprezentantul *t'* al mulţimii lui *t-1* şi se uneşte mulţimea lui *t* cu cea a lui *t-1*. Fiecare reprezentant menţine două câmpuri: *left* şi *right*. *left* reprezintă indicele primului moment de timp din interval şi *right* reprezintă indicele ultimului moment de timp din interval; la unirea lui *t* cu *t'*, *right[t']* devine *t*; dacă *t-1* nu este ocupat, atunci setăm *left[t]=right[t]=t*. Verificăm apoi dacă momentul *t+1* este ocupat (*t≤N-1*). Dacă da, găsim reprezentantul *t'* al mulţimii lui *t*, şi *t''* al mulţimii lui *t+1*. Vom decide care dintre cei doi reprezentanţi va fi reprezentantul mulţimii reunite conform euristicii folosite. Să presupunem că *t'''* va fi noul reprezentant. Vom seta *left[t''']=min{left[t'], left[t'']}* şi *right[t''']=max{right[t'], right[t'']}*.

Pentru a găsi cel mai mare moment de timp *t* mai mic sau egal ca *tf(i)*, verificăm dacă *tf(i)* este ocupat. Dacă nu, atunci *t=tf(i)*; altfel, determinăm *tr*=reprezentantul mulţimii din care face parte *tf(i)*; *t* va fi egal cu *left[tr]-1*. Dacă *t=0*, atunci nu există niciun moment de timp disponibil mai mic sau egal cu *tf(i)*. Complexitatea unei operaţii cu mulţimi disjuncte poate varia, în funcţie de implementarea aleasă, de la *O(log(N))* la *O(log*(N))*.

Problema 4-4. ATP (Lotul Naţional de Informatică, România 2002)

Primii N ($1 \le N \le 100.000$) jucători de tenis din clasamentul ATP participă la un turneu de tenis. Jucătorii sunt numerotaţi în funcţie de poziţia lor în clasament, cu numere de la *1* la *N* (jucătorul *i* este mai bun decât jucătorul *j*, dacă $i<j$). Se ştie că dacă doi jucători *i* şi *j* ($i<j$) joacă unul împotriva altuia în turneu, atunci *i* câştigă sigur dacă $|j-i|>K$ ($1 \le K \le N-1$). Dacă, în schimb, $|j-i| \le K$, atunci poate câştiga oricare dintre ei. Determinaţi care este cel mai slab jucător (cel mai jos în clasament) care ar putea câştiga turneul.

Se ştie că *N* este o putere a lui *2* şi că turneul se joacă în etape eliminatorii. În prima etapă cei *N* jucători se împart în *N/2* perechi care joacă unul împortiva celuilalt. Cei *N/2* câştigători avansează la etapa următoare, unde vor fi împărţiţi în *N/4* perechi ş.a.m.d. până se ajuge în finală (ultimii 2 jucători). Câştigătorul finalei este câştigătorul turneului.

Soluţie: Vom căuta binar (între *1* şi *N*) cel mai mare indice *x* al unui jucător care poate câştiga turneul. Pentru un *x* fixat, trebuie să verificăm dacă jucătorul *x* poate câştiga turneul. Dacă da, atunci vom testa în continuare jucători cu indici mai mari decât *x*; dacă nu, vom testa jucători cu indici mai mici decât *x*. Pentru a verifica dacă *x* poate câştiga finala, vom construi întreg tabloul de joc. În finală, cel mai bine ar fi ca *x* să joace cu cel mai bun jucător *y* pe care încă îl poate bate: $y=max\{x-K, 1\}$. Apoi va trebui să determinăm pentru *y* şi *x* cu cine să joace în semifinale. Pentru fiecare jucător *x'* dintre ei, în această ordine, vom alege acel jucător *y'* care nu este planificat să câştige la runda curentă şi care este cel mai bine plasat în clasament (*y'* se află în intervalul $[x'-K, N]$). Astfel, algoritmul se conturează în felul următor. Avem o listă de jucători $x'(1), ..., x'(Q)$ care trebuie să câştige meciul curent (ordonaţi astfel încât $x'(i)<x'(i+1)$, $1 \le i \le Q-1$), astfel că trebuie împerecheaţi cu un jucător pe care îl pot bate. Iniţial, avem doar jucătorul *x* ($x'(1)=x$). Pentru fiecare jucător *i* vom menţine starea lui (ocupat sau nu). Iniţial, doar *x* este marcat ca fiind ocupat.

Vom parcurge jucătorii $x'(j)$ (în ordine, $j=1,...,Q$) şi pentru fiecare vom căuta jucătorul liber $y'(j)$ cu indicele cel mai mic din intervalul $[max\{x'(j)-K, 1\}, N]$ şi vom împerechea jucătorii $x'(j)$ şi $y'(j)$. Dacă, la un moment dat, nu reuşim să găsim un jucător $y'(j)$ disponibil, atunci nu putem construi un tablou de joc astfel încât jucătorul iniţial *x* să câştige turneul. După determinarea unui jucăţor $y'(j)$, îl marcăm ca fiind ocupat. După determinarea tuturor jucătorilor $y'(j)$, vom interclasele listele $x'(*)$ şi $y'(*)$ (amândouă sunt sortate crescător după indicele jucătorilor) şi vom obţine lista $x'(*)$ cu $2 \cdot Q$ elemente ce va fi folosită la runda următoare. Algoritmul se termină cu succes atunci când lista *x'* îi conţine pe toţi cei *N* jucători.

Pentru determinarea jucătorului disponibil cu indicele cel mai mic dintr-un interval $[a,b]$, puem folosi una din tehnicile de la problema anterioară (acolo se căuta cel mai mare element disponibil dintr-un interval $[a,b]$, însă este evident că cele 2 probleme sunt interschimbabile – putem să privim şirul de poziţii din sensul celălalt şi, astfel, cerinţa de găsire a celui mai mic element disponibil dintr-un interval devine echivalentă cu cea de găsire a celui mai mare element disponibil dintr-un interval).

O soluţie alternativă mai „matematică” a fost prezentată de dl. prof. Stelian Ciurea. Descriem pe scurt soluţia sa în continuare. Să presupunem că notăm cu *a* jucătorul cel mai slab care ar putea câştiga turneul. Rezolvarea problemei are la bază următoarea afirmaţie: “Cea mai mare valoare a lui *a* se obţine dacă el întâlneşte în finală pe *a-K*.” Demonstraţia afirmaţiei se face prin reducere la absurd: presupunem că în finală *a* joacă cu alt jucător, fie

acesta *b*. *b* nu poate fi mai bine clasat decât *a-K* (în acest caz a nu l-ar putea învinge!). Deci *b>a-K*. Apar două situaţii:

- *a* îl întâlneşte pe *a-K* într-o etapă anterioară finalei. În acest caz, putem interschimba pe *a-K* cu *b*. Dacă *a-K* joacă la un moment dat cu un adversar pe care *b* nu îl poate invinge, interschimbăm şi respectivul adversar cu un adversar a lui *b* şi repetăm respectivul procedeu şi pentru adversarii adversarilor etc. (în cel mai rău caz, interschimbăm întreg subarborele corespunzător adversarului lui *a-K* pe care *b* nu îl poate invinge cu un subarbore al unui adversar al lui *b* de aceeaşi adâncime (evident vom găsi un astfel de subarbore, deoarece *b* se află pe un nivel superior lui *a-K*)
- *a* nu îl întâlneşte pe *a-K* într-o partidă directă. În acest caz, turneul ar putea fi câştigat de *a+1*, deoarece în arborele binar care reprezintă schema de desfăşurare a turneului în această situaţie, putem înlocui pe *a* cu *a+1*: *a+1* poate să învingă toţi jucătorii pe care îi poate invinge *a* cu exceţia lui *a-K*, iar cum *a* nu îl întâlneşte pe *a-K*, rezultă că putem face interschimbarea între *a* şi *a+1*.

Am ajuns deci la o contradicţie – o valoare mai mare pentru câştigătorul turneului, deci afirmaţia este adevărată. Putem face o afirmaţie asemănătoare şi pentru adversarul lui *a-K* din semifinală: acesta va fi $a\text{-}2\cdot K$, apoi pentru adversarul lui $a\text{-}2\cdot K$ în sfertul de finală, acesta fiind $a\text{-}3\cdot K$ ş.a.m.d. Putem să extindem afirmaţia şi mai mult, ajungând la concluzia că dacă completăm schema de desfăşurare a turneului începând de la finală, apoi semifinalele etc şi la un moment dat trebuie să alegem un adversar pentru un jucător oarecare *j*, atunci din adversarii "disponibili" pe care j îi poate învinge, cel mai avantajos este să îl alegem pe cel mai bine clasat în clasamentul ATP. Având în vedere schema de desfăşurare a turneului care rezultă din afirmaţiile de mai sus, putem să calculăm valoarea lui *a* în modul următor:

- *a* are nevoie de *x* adversari (*x*=numărul de tururi) pe care să-i învingă iar aceştia sunt
  - *a-K* în finală,
  - *a-K+1*, în semifinală
  - ....
  - *a-K+x-1* (dacă $K<x$) sau *a-K+x* dacă ($K\geq x$) în primul tur.

Rezultă $a-K+x\leq N$ (sau $a-K+x-1\leq N$), iar cum valoarea cea mai mare pentru a rezultă la egalitate, deducem $a = N + K - x$

- învinşii lui *a* au un număr de $x\cdot(x-1)/2$ adversari pe care îi înving: *a-K* joacă *x-1* tururi până să fie eliminat, *a-K+1* joacă *x-2* tururi, etc., iar aceşti adversari trebuie să fie cât mai bine clasaţi, primul dintre ei fiind $a\text{-}2\cdot K$.

Rezultă $a-2\cdot K + x + x\cdot (x-1)/2\leq N$ deci $a = N + 2\cdot K - x - x\cdot (x-1)/2$ sau notând cu $G(x-1) = x\cdot (x-1)/2$, $a = N + 2\cdot K - x - G(x-1)$

- învinşii învinşiilor lui *a* au un număr de adversari pe care îi înving egal cu $G(x-2) + G(x-3) + ... + G(2) + 1$ iar aceştia trebuie să fie cât mai bine clasaţi începând cu $a\text{-}3\cdot K$ rezultând
  $a = N + 3\cdot K - x - G(x-1) - G(x-2) - ... - G(2) - G(1)$

Se continuă, după un raţionament asemănător, calculul numărului de adversari de care au nevoie învinşii învinşilor învinşilor, ..., în calculul acestui număr observându-se că dacă în etapa precedentă a calculului a apărut un termen de genul $G(x-i)$, atunci în etapa curentă acesta generează o sumă de termeni $G(x-i-1) + G(x-i-2) + ... +G(2) + G(1)$.

Evident, din relaţiile obţinute se va reţine valoarea cea mai mare pentru *a*, aceasta fiind soluţia problemei. Completarea schemei de desfăşurare a turneului se va face apoi exact pe baza raţionamentului anterior. În program, pentru a evita calculul repetat al unor valori, se folosesc formule de genul

- $a_1 = N + K - x$
- $a_2 = a_1 + K - G(x\text{-}1)$
- $a_3 = a_2 + K - G(x\text{-}2)$ - ... $- G(2) - 1$
- $a_4 = a_3 + K - G(x\text{-}3) - 2{\cdot}G(x\text{-}4) - 3{\cdot}G(x\text{-}5)$
- ...

Pentru calculul coeficienţilor care apar la un moment dat în expresiile de mai sus se poate folosi o matrice (notată *m*) şi o procedură denumită "*derivare*" prin care se implementează observaţia că *G(x-i)* învinşi au nevoie de *G(x-i-1) + G(x-i-2) + ... +G(2) + G(1)* adversari pe care să-i întâlnească până în momentul când sunt la rândul lor învinşi.

Problema 4-5. Turnuri (SGU)

Gigel are în faţă *N* (*1≤N≤1.000.000*) celule iniţial goale. El se joacă cu aceste celule, adăugând sau eliminând cuburi din ele. Un turn este o secvenţă maximală de celule consecutive, cu proprietatea că orice celulă din secvenţă conţine cel puţin un cub. Turnurile pot fi numerotate de la stânga la dreapta (în ordinea în care apar), iar celulele din cadrul unui turn *[a,b]* pot fi numerotate de la *1* la *b-a+1*. Gigel efectuează operaţii de tipul următor:

- ***pune/elimină*** *x>0* cuburi în celula *y*;
- ***pune/elimină*** *x>0* cuburi în celula *y* a turnului *z*.

În plus, din când în când, Gigel îşi pune diverse întrebări, de următoarele tipuri:

(1) câte turnuri există? ;

(2) câte cuburi există în turnul *z* ? ;

(3) câte celule conţine turnul *z* ? ;

(4) câte cuburi conţine celula *y* din turnul *z* ?

Dându-se o secvenţă de *M* (*1≤M≤1.000.000*) operaţii şi întrebări, determinaţi răspunsul la fiecare întrebare (ţinând cont doar de operaţiile efectuate înaintea întrebării respective). Se garantează că numărul de cuburi din orice celulă nu va depăşi *2.000.000.000* şi nu va scădea sub *0*.

Soluţie: Vom menţine un vector *C*, unde *C(y)*=numărul de cuburi din celula *y* (*1≤y≤N*) (iniţial, *C(*)=0*). De asemenea, vom menţine un arbore echilibrat *T* ce va conţine intervalele corespunzătoare turnurilor existente la fiecare moment de timp (fiecare interval va avea şi o valoare asociată, reprezentând numărul de cuburi din interval). Vom menţine şi un contor *NT*, reprezentând numărul de intervale din *T* (deci, numărul de turnur existente). Iniţial, *NT=0* şi *T* nu conţine niciun interval. De fiecare dată când se adaugă un interval nou în *T*, incrementăm *NT* cu *1*; când ştergem un interval din *T*, decrementăm *NT* cu *1*. Astfel, am rezolvat problema determinării răspunsului la primul tip de întrebări.

Fiecare interval *[a,b]* din *T* va avea asociată şi o valoare *v*, reprezentând numărul de cuburi din celulele din cadrul intervalului.

O operaţie de tipul „pune *x* cuburi în celula *y*" se realizează după cum urmează. Dacă *C(y)=0*, atunci setăm *C(y)=x* şi efectuăm următorii paşi:

(1) introducem intervalul *[y,y]* în *T*, cu valoarea *x*;

(2) fie *a=b=y*;

(3) determinăm intervalul *[c,d]* din *T*, localizat imediat în stânga lui *[a,b]* ;

(4) dacă *[c,d]* există şi *d=a-1*, atunci eliminăm intervalele *[c,d]* şi *[a,b]* din *T* şi introducem în locul lor intervalul *[c,b]*, cu valoarea asociată egală cu suma valorilor asociate lui *[c,d]* şi *[a,b]*; în continuare, setăm *a=c*;

(5) determinăm intervalul *[e,f]* din *T*, localizat imediat în dreapta lui *[a,b]* ;

(6) dacă *[e,f]* există şi *e=b+1*, atunci eliminăm intervalele *[e,f]* şi *[a,b]* din *T* şi introducem în locul lor intervalul *[a,f]*, cu valoarea asociată egală cu suma valorilor asociate lui *[e,f]* şi *[a,b]*; în continuare, setăm *b=f*.

Dacă *C(y)>0* dinaintea operaţiei curente, atunci căutăm intervalul *[a,b]* din *T* care conţine celula *y* (determinăm acel interval *[a,b]* unde *a* este cea mai mare valoare mai mică sau egală cu *y*). Vom incrementa cu *x* atât *C(y)*, cât şi valoarea *v* asociată intervalului *[a,b]* din *T*.

O operaţie de tipul „pune *x* cuburi în celula *y* a turnului *z*" presupune determinarea intervalului *[a,b]* ce corespunde turnului *z*. Pentru aceasta, fiecare nod *q* din *T* conţine numărul de intervale *nint(q)* din subarborele său. Pornim din rădăcina arborelui, căutând al *z*-lea interval. Să presupunem că suntem în nodul *q* şi căutăm al *k*-lea interval din subarborele nodului *q*. Fie *qleft* şi *qright* fiii stânga şi dreapta ai lui *q*. Dacă *qleft* există şi *nint(qleft)≤k*, atunci continuăm căutarea celui de-al *k*-lea interval în *qleft*; dacă *qleft* există şi *nint(qleft)<k*, atunci decrementăm *k* cu *nint(qleft)*. Dacă nu am continuat căutarea în *qleft* şi *k=1*, atunci intervalul corespunzător nodului q este intervalul căutat. Dacă *k>1*, atunci decrementăm *k* cu *1* şi continuăm căutarea celui de-al *k*-lea interval în *qright* (*qright* sigur există, dacă indicele *z* iniţial este mai mic sau egal cu *NT*).

O dată ce am identificat intervalul *[a,b]* din *T*, determinăm celula *y'=a+y-1*. Incrementăm *C(y')* cu *x* şi mărim cu *x* şi valoarea asociată intervalului *[a,b]* în *T*.

Pentru o întrebare de tipul *(2)* determinăm intervalul *[a,b]* corespunzător celui de-al *z*-lea turn şi întoarcem valoarea asociată. Pentru o întrebare de tipul *(3)* determinăm intervalul *[a,b]* corespunzător celui de-al *z*-lea turn şi întoarcem *(b-a+1)*. Pentru întrebarea *(4)* determinăm celula (reală) *y'* ce corespunde celulei *y* din turnul *z* şi întoarcem *C(y')*.

Pentru operaţiile de eliminare vom proceda după cum urmează. Dacă nu se dă celula *y* relativă la un turn *z*, determinăm (ca mai sus) celula reală *y'* ce corespunde celulei *y* din turnul *z* şi setăm *y=y'*. În continuare, vom presupune că avem de-a face doar cu primul tip de operaţie de eliminare (în care se dă celula reală *y*).

Vom determina intervalul *[a,b]* din *T* ce conţine celula *y*. Dacă *C(y)>x*, atunci decrementăm cu *x* atât *C(y)*, cât şi valoarea *v* asociată intervalului *[a,b]* în *T*. Dacă *C(y)=x*, atunci turnul *[a,b]* va fi impărţit în (cel mult) alte două intervale *[a,y-1]* (dacă *a≤y-1*) şi *[y+1,b]* (dacă *y+1≤b*). Vom elimina din *T* intervalul *[a,b]*. Să presupunem acum că *[p,q]* (*p≤q*) este unul din cele (cel mult) două intervale în care a fost împărţit intervalul *[a,b]*. Va trebui să determinăm numărul total de cuburi din intervalul de celule *[p,q]*. Pentru aceasta, putem menţine separat un arbore de intervale peste cele *N* celule, în care putem actualiza o valoare *C(y)* şi putem calcula suma valorilor dintr-un interval în timp *O(log(N))*. Folosind acest arbore de intervale, determinăm valoarea *v* ce va fi asociată intervalului *[p,q]* şi introducem intervalul *[p,q]* în *T*. Procedăm la fel şi pentru celălalt interval care a rezultat din împărţirea lui *[a,b]* (dacă există).

Complexitatea fiecărei operaţii este *O(log(N))*.

Problema 4-6. Cea mai apropiată sumă (UVA)

Se dau două secvenţe de *M*, respectiv *N* (*1≤M, N≤100.000*) numere întregi ordonate crescător. Prima secvenţă conţine numerele *a(i)* (*1≤i≤M*; *a(i)≤a(i+1)*), iar cea de-a doua numerele *b(j)* (*1≤j≤N*; *b(j)≤b(j+1)*). Dându-se o valoare *S*, determinaţi perechea de numere *(i,j)* cu proprietatea că |*a(i)+b(j)-S*| este minim.

Soluţie: O primă soluţie este următoarea. Vom considera, pe rând, fiecare valoare *a(i)*. Apoi vom căuta binar în a doua secvenţă cea mai mare valoare *b(j)*, mai mică sau egală cu *S-a(i)*

(dacă există; altfel setăm $j=0$). Vom calcula apoi $S_1(i)=a(i)+b(j)$ (dacă $j \geq 1$) şi $S_2(i)=a(i)+b(j+1)$ (dacă $j \leq N-1$). Cea mai mică valoarea a modulului este $min\{|S_1(i)-S|, |S_2(i)-S| \mid 1 \leq i \leq N\}$. Complexitatea acestei soluţii este $O(N \cdot log(N))$.

Voi prezenta acum o a doua soluţie, ce are complexitate liniară. Vom determina în timp $O(N)$ indicele j pentru fiecare valoare $a(i)$. După determinarea acestui indice, algoritmul calculează sumele $S_1(i)$ şi $S_2(i)$ şi determină diferenţa în modul minimă (ca mai sus). Vom începe cu $i=0$ şi $j=N$. Vom incrementa apoi, pe rând, valoarea lui i (de la 1 la N). Pentru fiecare valoare a lui i procedăm după cum urmează: cât timp ($j \geq 1$) şi ($a(i)+b(j)>S$), decrementăm valoarea lui j cu 1. Valoarea lui j de la sfârşitul acestui ciclu este indicele j corespunzător valorii $a(i)$. Este evident că indicele i doar creşte şi indicele j doar scade, astfel că obţinem complexitatea liniară dorită.

Capitolul 5. Şiruri de Caractere

Problema 5-1. Base3 (Olimpiada Naţională de Informatică, 2004, cls. 11-12, România)

Se dau *3* numere scrise în baza *3* (folosind cifrele *0*, *1* şi *2*). Se doreşte găsirea unui număr *N* în baza *3*, care să aibă un număr impar de cifre, iar cifra de pe poziţia din mijloc să aibă valoarea *1*. Acest număr *N* trebuie obţinut prin concatenarea celor trei numere date; în această concatenare, fiecare din cele *3* numere poate fi folosit de zero sau mai multe ori. Determinaţi numărul minim de cifre pe care îl poate avea un număr având proprietăţile precizate.

Numărul de cifre al fiecăruia din cele *3* numere este un număr întreg între *1* şi *16000*. Numerele date pot conţine zerouri la început; acestea trebuie luate în considerare, dacă numărul respectiv este folosit în concatenare.

Exemplu:

001 020 2020	Se poate obţine un număr având numărul minim de cifre 13: *2020***001***001***001**.

Soluţie: Se calculează matricea *MIN[i, x ,j]*, cu *i* între *1* şi *3*, *j* între *1* şi *2*, iar *x* între *0* şi lungimea numărului *i*, având următoarea semnificaţie :

- dacă *j=1*, atunci *MIN[i, x, j]* reprezintă lungimea celui mai scurt număr care are la mijloc primele *x* cifre din al *i*-lea număr
- dacă *j=2*, atunci *MIN[i, x, j]* reprezintă lungimea celui mai scurt număr care are la mijloc ultimele *x* cifre din al *i*-lea număr

Calculul acestor valori corespunde unei determinări a numărului din exterior spre centru. Cei *3* indici ai matricei codifică o stare, iar trecerea de la o stare la alta se realizează prin concatenarea unuia din numere în partea stângă sau dreaptă. Astfel, se poate folosi un algoritm de drum minim (de exemplu, Dijkstra cu heap-uri). Conform acestei codificări, numărul cerut corespunde unui drum minim în graful stărilor, iar lungimea acestui drum este limitată superior de $\mathbf{6 * 16000^2}$ (dar, în practică, este mai mică).

Complexitatea algoritmului este $O((L_1 + L_2 + L_3)\cdot log(L_1 + L_2 + L_3))$ (L_i=numărul de cifre al celui de-al *i*-lea număr, $1 \le i \le 3$).

Problema 5-2. Sub (Lotul Naţional de Informatică, România, 2008)

Fie *A* şi *B* două mulţimi de şiruri formate doar din litere mici ale alfabetului englez (de la *„a"* la *„z"*). Fie *Na* ($1 \le Na \le 100$) numărul şirurilor din mulţimea *A*, iar *Nb* ($1 \le Nb \le 100$) numărul şirurilor din mulţimea *B*. Se spune că *s(1)s(2)...s(k)* este o subsecvenţă a unui şir *a(1)a(2)...a(n)* dacă există un număr natural *i* ($1 \le i \le n-k$) astfel încât *s(1)=a(i), s(2)=a(i+1), ... s(k)=a(i+k-1)*.

Determinaţi numărul şirurilor care sunt subsecvenţe ale tuturor şirurilor din *A*, dar nu sunt subsecvenţe ale niciunui şir din *B*. Lungimea oricărui şir din cele două mulţimi nu depăşeşte *300*.

Exemplu: Mulţimea *A* este *{„abcaf", „bcaf", „dbdafabcaf"}*, iar mulţimea *B* este *{„bacbc", „fbca", „ca"}*. Şirurile *„a"*, *„b"*, *„c"*, *„f"*, *„bc"*, *„ca"*, *„af"*, *„bca"*, *„bcaf"*, *„caf"* sunt subsecvenţe ale tuturor şirurilor din A. Dintre acestea, şirurile: *„a"*, *„b"*, *„c"*, *„f"*, *„ca"*, *„af"*, *„bca"* sunt subsecvenţe ale cel puţin unui şir din *B*. Rămân *3* şiruri care sunt

subsecvenţe ale tuturor şirurilor din *A*, dar nu sunt subsecvenţe ale niciunui şir din *B*: *„af”*, *„caf”*, *„bcaf”*.

Soluţie: Vom construi un trie (arbore de prefixe) ce va conţine toate subsecvenţele şirurilor din *A*. Fiecare nod *x* al trie-ului va stoca numărul de şiruri din mulţimea *A* din care face parte şirul *S(x)* corespunzător nodului *x* (*S(x)* se obţine prin concatenarea etichetelor muchiilor de pe drumul de la rădăcină până la nodul *x*), *na(x)*, indicele ultimului şir din *A* care a introdus *S(x)* în trie (sau al ultimului şir din *B* în urma căruia s-a trecut prin nodul *x*), *last(x)*, şi o valoare *nb(x)*, reprezentând numărul de şiruri din *B* ce conţin *S(x)* ca subsecvenţă.

Vom porni cu un trie gol (conţine doar nodul rădăcină). Vom parcurge cuvintele din mulţimea *A* în ordine crescătoare a indicelui lor *i* ($1 \le i \le Na$). Pentru fiecare cuvânt *CA(i)* considerăm fiecare poziţie *j* a sa ($1 \le j \le |CA(i)|$; $|CA(i)|$=lungimea lui *CA(i)*). Introducem în trie subsecvenţa lui *CA(i)* care începe la poziţia *j* (şi se termină la ultimul caracter al lui *CA(i)*). Pe măsură ce parcurgem această subsecvenţă cu un indice *k* ($j \le k \le |CA(i)|$), va trebui să introducem noduri noi sau doar să coborâm într-unul din fiii nodului curent.

Să presupunem că, după parcurgerea caracterului *CA(i,k)* suntem la nodul *x* în trie. Dacă $last(x) \ne i$, atunci incrementăm *na(x)* cu *1* (*last(x)*, *na(x)* şi *nb(x)* sunt iniţializate la *0*, la crearea nodului) şi setăm *last(x)=i*. După construcţia trie-ului, resetăm valorile *last(x)* la *0*.

Vom parcurge acum şirurile din mulţimea *B*, în ordine crescătoare a indicelui lor *i* ($1 \le i \le Nb$). La fel ca şi în cazul cuvintelor din mulţimea *B*, considerăm fiecare cuvânt *CB(i)* şi, pentru fiecare astfel de cuvânt, considerăm fiecare poziţie *j* ($1 \le j \le |CB(i)|$) din el. Începem să inserăm în trie subsecvenţa ce începe la poziţia *j* în *CB(i)* şi se termină pe ultima poziţie a acestuia. Pe măsură ce în cadrul subsecvenţei înaintăm cu un indice *k* ($j \le k \le |CB(i)|$), în cadrul trie-ului coborâm din nodul curent *x* (iniţial *x* este rădăcina) într-un fiu al acestuia. Să presupunem că tocmai am ajuns la poziţia *k* din subsecvenţa curentă *CB(i)* ($j \le k$). Dacă *x* nu are o muchie etichetată cu *CB(i,k)* către un fiu *F* de-al său, atunci creăm acest fiu *F* şi îl setăm pe *x* la valoarea lui *F* (iniţializăm *na(x)=nb(x)=last(x)=0*); altfel, setăm valoarea *x* a nodului curent la fiul *F*. Dacă $last(x) \ne i$, atunci incrementăm *nb(x)* cu *1* şi setăm *last(x)=i*.

Pentru a da răspunsul la problemă, la final numărăm câte noduri *x* ale trie-ului au *na(x)=Na* şi *nb(x)=0*. Observăm că, prin metoda folosită, am rezolvat mai multe probleme, de fapt. De exemplu, putem determina câte subsecvenţe sunt subsecvenţe ale cel puţin *XA* şi cel mult *YA* şiruri din *A*, precum şi ale cel puţin *XB* şi cel mult *YB* şiruri din *B*.

Pentru problema noastră putem optimiza puţin consumul de memorie. Când parcurgem şirurile din mulţimea *B*, dacă trebuie să creăm un nod nou, ne oprim cu parcurgerea subsecvenţei curente şi trecem la subsecvenţa următoare din cadrul aceluiaşi şir (sau la şirul următor, dacă era ultima subsecvenţă de considerat).

Complexitatea algoritmului este $O((Na+Nb) \cdot LMAX^2)$, unde *LMAX* reprezintă lungimea maximă a unui şir din cele *2* mulţimi.

Problema 5-3. Şiruri (Stelele Informaticii 2005)

Se dau două şiruri de numere *X* şi *Y*, având între *1* şi *100.000* elemente. Determinaţi o subsecvenţă de lungime maximă de elemente aflate pe poziţii consecutive, astfel încât: $X(p)+Y(q)=X(p+1)+Y(q+1)=...=X(p+L-1)+Y(q+L-1)$, unde *L* este lungimea secvenţei (am notat prin *A(i)* al *i*-lea element din şirul *A*).

Soluţie: Vom construi un şir *Y'* ce conţine elementele din *Y* negate: $Y'(i)=-Y(i)$ ($1 \le i \le |Y|$). Acum trebuie să determinăm o subsecvenţă de lungime maximă *L* astfel încât $X(p)-Y'(q)=X(p+1)-Y'(q+1)=...=X(p+L-1)-Y'(q+L-1)$.

Vom construi şirurile X_1 şi Y_1, având $|X|-1$, respectiv $|Y|-1$ elemente, unde $X_1(i)=X(i+1)-X(i)$ ($1 \le i \le |X|-1$) şi $Y_1(i)=Y'(i+1)-Y'(i)$ ($1 \le i \le |Y|-1$). Observăm că determinarea unei subsecvenţe de lungime maximă cu proprietăţile cerute este echivalentă cu determinarea celei mai lungi subsecvenţe comune a şirurilor X_1 şi Y_1. Fie *L* lungimea aceste subsecvenţe comune ($X_1(p+i)=Y_1(q+i)$, $0 \le i \le L-1$). Atunci există o subsecvenţă de lungime $L+1$ care începe la poziţia *p* în şirul *X* şi poziţia *q* în şirul *Y* şi care are proprietăţile cerute.

În continuare ne vom concentra pe determinarea lungimii maxime a unei subsecvenţe (continue) comune a două şiruri *A* şi *B*. Putem realiza acest lucru folosind programare dinamică. Calculăm *Lmax(i,j)*=lungimea maximă a subsecvenţei comune a şirurilor formate din primele *i* caractere din şirul *A* şi primele *j* caractere din şirul *B*, care se termină pe poziţiile *i*, respectiv *j*. Avem $Lmax(0,*)=Lmax(*,0)=0$. Dacă $A(i)=B(j)$ ($1 \le i \le |A|$, $1 \le j \le |B|$), atunci $Lmax(i,j)=1+Lmax(i-1, j-1)$; altfel, $Lmax(i,j)=0$. Lungimea subsecvenţei este egală cu $max\{Lmax(i,j)\}$.

Această abordare se poate generaliza pentru cazul în care avem asociată o pondere pentru fiecare element din şirul *A*, respectiv *B* (ponderile $wA(i) \ge 0$ şi $wB(j) \ge 0$) şi dorim să găsim o subsecvenţă având o pondere "combinată" maximă. În acest caz, avem nevoie de două funcţii *f* şi *g* cu valori nenegative, care "combină" ponderile a două poziţii care se "potrivesc" (*i* şi *j*), respectiv "combină" valorile pentru toate poziţiile din subsecvenţă. În acest caz, relaţiile devin: dacă $A(i) \ne B(j)$, atunci $Lmax(i,j)=0$; altfel, $Lmax(i,j)=max\{g(f(wA(i), wB(j)), Lmax(i-1, j-1)), f(wA(i), wB(j)), 0\}$.

Totuşi, algoritmul descris mai sus are complexitatea $O(|A| \cdot |B|)$, care este prea mare pentru limitele problemei. Pentru determinarea subsecvenţei (continue) comune de lungime maximă vom construi şirul *Z=A$B*, format din şirul *A*, urmat de un element *$* şi apoi urmat de şirul *B*. Elementul *$* este un element care nu apare în niciunul din şirurile *A* şi *B*. Vom construi apoi şirul de sufixe corespunzător şirului *Z*, adică, practic, se sortează lexicografic sufixele şirului *Z*, în ordine: $s(1), s(2), ..., s(|Z|)$. Pentru fiecare sufix *s(i)* putem şti exact din ce şir (*A* sau *B*) face parte primul său element (cunoaştem poziţia sa în şirul *Z* şi determinăm dacă se află înaintea elementului *$* sau după acesta); sufixul care începe cu caracterul *$* nu ne va interesa.

Apoi, folosind informaţiile păstrate la construcţia şirului de sufixe, putem determina şirul *LCP*, unde *LCP(i)*=cel mai lung prefix comun al sufixelor *s(i)* şi *s(i+1)* ($1 \le i \le |Z|-1$). Putem construi şirul de sufixe în timp $O(|Z| \cdot log(|Z|))$ şi apoi putem calcula *LCP(i)* în timp $O(log(|Z|))$ pentru fiecare poziţie *i*.

Lungimea *L* maximă a unei subsecvenţe continue a şirurilor *A* şi *B* este *max{LCP(i)*| *s(i)* are primul caracter într-unul din şiruri (*A* sau *B*), iar *s(i+1)* are primul caracter în celălalt şir (*B* sau *A*)*}*.

Problema 5-4. Verificarea unei secvenţe (TIMUS)

Se dă o secvenţă de numere $A(1), ..., A(N)$ ($1 \le N \le 100.000$). Determinaţi dacă orice subsecvenţă $A(i), ..., A(j)$ ($1 \le i \le j \le N$) verifică următoarea proprietate: $A(i)+A(i+1)+...+A(j) \le P \cdot (j-i+1)+Q$.

Soluţie: Vom construi şirul $B(i)=A(i)-P$. Observăm că dacă orice subsecvenţă din acest şir, $B(i), ..., B(j)$ verifică proprietatea: $B(i)+...+B(j) \le Q$, atunci orice subsecvenţă a şirului *A*

verifică proprietatea din enunţul problemei. Verificarea proprietăţii şirului *B* se poate realiza foarte simplu. Determinăm (în timp *O(N)*, folosind unul din mulţi algoritmi standard existenţi) subsecvenţa de sumă maximă din *B*. Fie *S* suma aceste subsecvenţe. Dacă $S \leq Q$, atunci orice subsecvenţă verifică proprietatea; altfel, nu.

Problema 5-5. Expoziţie (SGU)

Două companii, *A* şi *B*, vor să îşi expună produsele la o expoziţie. Produsele sunt expuse în standuri localizate unul după altul, în linie (de la stânga la dreapta). Fiecare stand poate fi gol sau plin. Standurile goale sunt etichetate cu numele uneia din cele două companii. La orice moment de timp, oricare din cele două companii poate realiza una din următoarele două acţiuni:

(1) compania poate alege oricare din standurile goale etichetate cu numele său şi poate amplasa acolo un produs al celeilalte companii ;

(2) compania poate alege un stand gol etichetat cu numele său, apoi: *(i)* amplasează în stânga acestui stand două standuri noi, după cum urmează: un stand cu un produs propriu (un stand fără etichetă) şi un stand gol etichetat cu numele celeilalte companii.

Ştiind că iniţial exista un singur stand gol etichetat cu numele uneia din cele două companii (numele este cunoscut) şi că la final s-au umplut cu produse toate standurile existente, determinaţi dacă şirul final de standuri umplute (pentru fiecare stand, de la stânga la dreapta, se cunoaşte compania al cărei produs este amplasat în stand) poate fi obţinut din starea iniţială prin cele două tipuri de acţiuni descrise.

Exemplu: Iniţial există un stand gol etichetat cu *A*. La final există *3* standuri, conţinând, în ordine, produse ale companiilor *A*, *A* şi *B*. Şirul final poate fi obţinut din starea iniţială.

Soluţie: Problema poate fi scrisă prin definirea unei gramatici independente de context. Vom considera un alfabet format din simbolurile *{A, B, NA* şi *NB}*. *NA* şi *NB* reprezită standuri goale etichetate cu numele companiei *A*, respectiv *B*; *A* şi *B* reprezintă standuri pline, conţinând un produs al companiei *A*, respectiv *B*. Şirul iniţial conţine caracterul *NA* sau *NB*.

Avem *2* reguli de gramatică:

(1) NA->B | A NB NA şi

(2) NB->A | B NA NB.

X -> Y | Z are semnificaţia că (caracterul) *X* poate fi înlocuit cu şirurile *Y* sau *Z* (care pot conţine *0*, *1* sau mai multe caractere). Problema care se pune este dacă şirul dat poate fi obţinut din şirul iniţial folosind aceste reguli de gramatică. Pentru aceasta, vom folosi o stivă *S*. Iniţial, vom pune în stivă caracterul iniţial dat (*NA* sau *NB*). Vom parcurge apoi şirul de la stânga la dreapta. Dacă în şir avem caracterul *A* (*B*) şi în vârful stivei avem caracterul *NB* (*NA*), atunci eliminăm caracterul din vârful stivei. Dacă în şir avem caracterul *A* (*B*) şi în stivă avem caracterul *NA* (*NB*), atunci adăugăm în vârful stivei caracterul *NB* (*NA*). La final, stiva trebuie să fie goală. Dacă stiva nu este goală, atunci şirul dat nu poate fi obţinut din şirul (caracterul) iniţial.

Problema 5-6. abc (Happy Coding 2007, infoarena)

Alfabetul limbii Makako este compus din numai *3* simboluri: *a*, *b* şi *c*. Orice cuvânt al acestei limbi este un şir format dintr-un număr finit de simboluri din alfabet (la fel ca în cele mai multe din limbile folosite în prezent). Totuşi, nu orice înşiruire de simboluri formează un cuvânt cu sens. Conform dicţionarului limbii Makako, numai *N* ($1 \leq N \leq 50.000$) şiruri de simboluri reprezintă cuvinte cu sens (în continuare, prin cuvânt vom înţelege unul dintre

aceste şiruri de simboluri ce au sens). O particularitate a limbii Makako este că oricare două cuvinte au exact aceeaşi lungime *LC* (între *1* şi *20*).

De curând s-a descoperit un text antic despre care se presupune că ar fi scris într-un dialect vechi al limbii Makako. Pentru a verifica această ipoteză, oamenii de ştiinţă vor să determine în ce poziţii din text se regăsesc cuvinte din limbă. Textul poate fi privit ca o înşiruire de *L* ($1 \le L \le 10.000.000$) simboluri din alfabetul limbii Makako, în care poziţiile simbolurilor sunt numerotate de la *1* la *L*. Dacă un cuvânt din limbă se regăseşte ca o înşiruire continuă de simboluri în cadrul textului, iar poziţia de început a acestuia este *P*, atunci *P* reprezintă o poziţie candidat. Oamenii de ştiinţă doresc să determine numărul poziţiilor candidat din cadrul textului.

Să presupunem că dicţionarul limbii Makako ar conţine doar următoarele *3* cuvinte: *bcc*, *aba* si *cba*, iar textul antic descoperit ar fi *cabababcba*. La poziţiile *2* şi *4* din text se regăseşte cuvântul *aba*. La poziţia *7* se regăseşte cuvântul *cba*. Cuvântul *bcc* nu se regăseşte în text. Aşadar, în text există *3* poziţii candidat.

Soluţie: O primă abordare care ar părea să aibă şanse de a obţine punctaj maxim este de a construi un trie al cuvintelor. Cu ajutorul acestui trie putem verifica în complexitate *O(L)* dacă există vreun cuvânt care să înceapă la fiecare poziţie *i* din sir. Totuşi, această solutie are complexitate $O(LC \cdot L)$ şi nu s-ar încadra în timp.

Optimizarea constă în construirea automatului finit determinist asociat celor *N* cuvinte. Acest automat poate fi construit in complexitate $O(N \cdot L)$, conform algoritmului Aho-Corasick. Modul de constructie este asemănător calculului funcţiei prefix din algoritmul KMP. Construim trie-ul cuvintelor, apoi, pentru un nod *X* al trie-ului, determinăm nodul din trie care corespunde celui mai lung şir de caractere care este sufix al şirului asociat nodului *X*. Putem calcula aceste valori pe nivele, întrucât un nod *Y* de tipul celui menţionat mai sus se va afla întotdeauna pe un nivel superior nodului *X* (dar nu neapărat pe calea de la rădăcina trie-ului până la nodul *X*). Acest automat se poate folosi apoi în cadrul parcurgerii textului dat.

Începem din starea corespunzătoare şirului vid (rădăcina trie-ului). De fiecare dată când trecem la următorul caracter, încercăm să trecem în nodul din trie ce corespunde caracterului respectiv şi este fiu al starii curente. Dacă acest nod nu există, la fel ca la KMP, trecem în starea corespunzatoare „prefixului-sufix" al stării curente şi repetăm acest lucru până când ajungem în rădăcina trie-ului sau într-un nod care are un fiu ce corespunde caracterului următor din şir (o variantă similară a acestui algoritm se foloseşte şi pentru calculul funcţiilor prefix corespunzătoare fiecărui nod). Dacă ajungem într-un nod ce corespunde unei stări finale, atunci am găsit o apariţie a unui cuvânt.

O altă soluţie constă în construcţia unui automat finit determinist, avand complexitate $O(N \cdot L^2)$. Se va construi trie-ul cuvintelor, apoi, pe baza acestui trie, se va calcula un automat care are un număr de stări egal cu numărul de noduri ale trie-ului. Pentru fiecare stare *X* şi fiecare caracter *c*, se va calcula funcţia *nextState[X,c]*, reprezentând starea ce corespunde celui mai lung şir care este un subşir al şirului corespunzător stării *X*, concatenat cu caracterul *c*. Pentru stările *X* ce au fiu în trie corespunzător unui caracter *c*, *nextState[X,c]* va fi egal cu acest fiu.

Pentru celelalte caractere şi o stare *X*, vom avea *O(L)* stări candidate. Mai exact, să considerăm că *TX* este tatăl nodului *X* în trie (presupunând că *X* nu este rădăcina trie-ului) şi că avem stările $Q_0, Q_1, .., Q_P$ stări ce corespund sufixelor de lungime *0, 1, ..., P* ale şirului corespunzător nodului *TX* (acest şir are *P+1* caractere). Atunci mulţimea de stări $R_1, ..., R_{P+1}$

având aceeaşi semnificaţie pentru nodul *X* se calculează ca fiind R_1=*fiu[Q_0,u]*, R_2=*fiu[Q_1,u]*, ..., R_{P+1}=*fiu[Q_P,u]*, unde *X=fiu[TX,u]*, adică fiul nodului *TX*, ce corespunde caracterului *u*. La această mulţime vom adăuga pe R_0=radacina trie-ului (ce corespunde şirului vid). *nextState[X,c]* va fi egal cu şirul de lungime maximă dintre şirurile corespunzătoare nodurilor *fiu[R_i,c]* (dacă există), cu $0 \le i \le P+1$ (sau R_0, dacă niciunul din nodurile R_p nu are un fiu corespunzător caracterului *c*).

Se observă uşor că pentru orice nod *X* pot exista cel mult *O(L)* stări candidate, deoarece şirul corespunzător unui nod *X* are *O(L)* caractere. Observăm că funcţia *nextState[X,c]* poate fi definită şi pe baza valorilor calculate de algoritmul Aho-Corasick, ea putând fi tabelată (precalculată pentru toate perechile *(X,c)*) sau calculată atunci când este nevoie de ea.

O soluţie mai simplă constă în folosirea unor funcţii de hash, ca în algoritmul Rabin-Karp. Aceste funcţii trebuie să fie calculabile uşor (în timp *O(1)*) atunci când avem valoarea hash-ului pentru un şir *S* şi dorim să calculăm valoarea pentru şirul *S'* obţinut prin adăugarea unui caracter la dreapta lui *S* şi înlăturarea unui caracter din stânga lui *S*. Câteva variante de functii de hash ar putea fi următoarele:

- funcţii polinomiale, cu puteri de numere prime: $(c_N \cdot P^N + c_{N-1} \cdot P^{N-1} + c_1 \cdot P + c_0)$ *mod Q* (*P* şi *Q* fiind numere prime)
- scrierea şirurilor ca numere în baza *3* (întrucât alfabetul constă doar din *3* litere) => este ideea de mai sus, dar cu *P=3*.

Problema 5-7. Cel mai scurt subşir al lui A care nu este subşir al lui B (Olimpiada Naţională de Informatică, Croaţia 2003)

Se dau *2* şiruri, *A* şi *B*, având cel puţin *1* şi cel mult *3.000* de caractere din mulţimea *{1,...,K}* ($1 \le K \le 3.000$). Determinaţi lungimea celui mai scurt subşir al lui *A* care nu este subşir al lui *B*. Un şir *P* este subşir al lui *Q* dacă se poate obţine din *Q* prin ştergerea a *0* sau mai multe caractere (deci, caracterele din *P* nu trebuie să se afle pe poziţii consecutive în *Q*).

Soluţie: Fie *len(A)* şi *len(B)* lungimile şirurilor *A* şi *B*. Vom nota prin *A(i)* (*B(j)*) al *i*-lea (*j*-lea) caracter din şirul *A* (*B*) (vom considera poziţiile numerotate de la *1*). Pentru şirul *B* vom calcula valorile *Prev(i,c)*=poziţia *j* ($1 \le j \le i$) cea mai mare pe care se află caracterul *c* între primele *i* caractere ale şirului *B* (sau *0* dacă acest caracter nu se regăseşte printre primele *i* caractere ale lui *B*). Avem *Prev(0,c)=0* ($1 \le c \le K$). Pentru $1 \le i \le len(B)$, vom avea: *Prev(i,B(i))=i* şi $Prev(i,c \neq B(i))=Prev(i-1,c)$.

În continuare vom folosi metoda programării dinamice. Vom calcula valorile *Lmin(i,j)*=lungimea minimă a unui subşir al primelor *i* caractere ale şirului *A* care nu este subşir al primelor *j* caractere ale şirului *B*. Vom avea $Lmin(0,0 \le j \le len(B))=+\infty$. Pentru $1 \le i \le len(A)$ avem:

(1) Lmin(i,0)=1;

(2) pentru $1 \le j \le len(B)$, dacă *Prev(j,A(i))=0*, atunci *Lmin(i,j)=1*; altfel, avem 2 cazuri:

(a) considerăm că subşirul determinat nu se termină la poziţia *i* din *A*; şi

(b) considerăm că subşirul determinat se termină la poziţia *i* din *A*.

De aici rezultă *Lmin(i,j)=min{Lmin(i-1,j), 1+Lmin(i-1, Prev(j, A(i))-1)}*. Cazul *(a)* corespunde termenului *Lmin(i-1,j)* (este ca şi cum nu am avea *i* poziţii, ci *i-1*). Cazul *(b)* corespunde termenului *1+Lmin(i-1, Prev(j, A(i))-1)*.

Determinăm un subşir al primelor *i-1* caractere ale lui *A* care nu este subşir al primelor *Prev(j, A(i))-1* caractere din şirul *B*, la care adăugăm (la sfârşit) caracterul *A(i)*. Subşirul

obţinut este un subşir al primelor i caractere din A, dar nu este un subşir al primelor j caractere din B. Complexitatea acestui algoritm este $O((len(A)+K)\cdot len(B))$.

Problema 5-8. Tunes (CPSPC 2004)

Considerăm un alfabet ce conţine n simboluri ($1 \le n \le 50.000$) numerotate de la 0 la $n-1$. Un şir $s_1, ..., s_k$ compus din caractere ale acestui alfabet se numeşte *bun* dacă pentru orice valori i şi j ($1 \le i \le k-1$ şi $i+2 \cdot j-1 \le k$), şirurile $s_i, ..., s_{i+j-1}$ şi $s_{i+j}, ..., s_{i+2 \cdot j-1}$ nu conţin exact aceleaşi caractere (adică există cel puţin un caracter x care apare de un număr diferit de ori în cele două şiruri).

Un şir *bun* se numeşte *neextensibil* dacă nu se poate obţine un alt şir bun prin adăugarea a oricărui număr caractere la începutul şi/sau la sfârşitul acestuia. Determinaţi un şir bun şi neextensibil de lungime minimă, care conţine fiecare caracter al alfabetului cel puţin o dată.

Soluţie: Pentru $n=1$ singurul şir bun este 0, iar pentru $n=2$ există 2 şiruri bune şi neextensibile: $0,1,0$ şi $1,0,1$. Pentru $n=3$, un şir bun şi neextensibil de lungime minimă este: $0, 1, 0, 2, 1, 2$.

Pentru $n \ge 4$, o primă soluţie este următoarea. Definim funcţiile $u(a,b)$ şi $v(a,b)$, care întorc şiruri de caractere. $u(a,b)$ întoarce şirul de caractere $a, a+1, ..., b$ (care conţine caracterele de la a la b, în ordine crescătoare). $v(a,b)$ întoarce şirul $a, a-1, a+1, a, ..., b, b-1$ (deci şirul ce constă din concatenarea şirurilor k, $k-1$, în ordine, cu k de la a la b). Un şir bun neextensibil de lungime minimă este următorul: $0, v(1,n-4), n-3, n-1, n-4, n-2, u(3,n-2), 0, n-1, u(1,n-4), 1, 3, 0, 2, v(4,n-1), n-1$.

O altă soluţie este următoarea. Fie funcţia $q(a)$=dacă a este impar atunci $((a+3)$ *div* $2)$ altfel $a/2$. Definim apoi funcţia $f(a)$, care întoarce un şir de caractere: $q(1), q(2), ..., q(a)$ (deci şirul format din caracterele $q(i)$, în ordine, pentru i de la 1 la a). O soluţie pentru problema noastră este următoarea: $0, u(2,n-2), u(1,n-2), 0, f(n-2), 0, u(2,n-1), u(2,n-2), 0$. u este aceaşi funcţie folosită în soluţia anterioară.

Problema 5-9. Şiruri Fibonacci (ACM SEERC 1997, enunţ modificat)

Să considerăm şirurile $F(0)$=”” (şirul vid), $F(1)$=”A”, $F(2)$=”B” şi $F(i \ge 3)=concatenare(F(p(i,1)), ..., F(p(i,NK(i))))$ (obţinut prin concantenarea şirurilor $F(p(i,1)), ..., F(p(i,NK(i)))$, în ordine, unde $p(i,j)<i$). De exemplu, putem avea $NK(i)=2$ şi $p(i,1)=i-1$ şi $p(i,2)=i-2$ (pentru $i \ge 3$).

Se dă şi un şir S (de lungime cel mult 20). Determinaţi de câte ori apare S în şirul $F(N)$ ($1 \le N \le 30.000$).

Soluţie: Vom genera şirurile $F(i)$ ($i \ge 1$) conform regulii de concatenare, până când ajungem la $i=N$ sau până când $len(F(i)) \ge len(S)$ ($len(X)$ reprezintă lungimea şirului X). Să presupunem că am ajuns la şirul $F(k)$ ($k \le N$). Vom calcula prin orice metodă de câte ori apare S în fiecare şir $F(i \le k)$ (de exemplu, încercăm fiecare poziţie de start şi verificăm dacă S se potriveşte începând de la acea poziţie). Fie $NA(i)$=numărul de apariţii ale lui S în $F(i)$. Avem calculate până acum toate valorile $NA(i \le k)$.

Fie $pref(i)$=prefixul de lungime (cel mult) $len(S)-1$ al şirului $F(i)$ şi $suf(i)$=sufixul de lungime (cel mult) $len(S)-1$ al şirului $F(i)$. Vom avea $pref(i \le k-1)=suf(k-1)=F(k-1)$, iar $pref(k)$ şi $suf(k)$ sunt obţinute prin păstrarea primelor (respectiv ultimelor) $len(S)-1$ caractere din $F(k)$. Dacă $k<N$, atunci calculăm şi F(k+1), pref(k+1), suf(k+1) şi NA(k+1) (tot prin metoda forţei brute), incrementând după aceea pe k cu 1.

Apoi, cât timp $k<N$, vom proceda după cum urmează: incrementăm pe k cu 1; apoi setăm $NA(k)$=suma valorilor $NA(p(k,j))$ ($1 \leq j \leq NK(k)$). Construim apoi $pref(k)$ şi $suf(k)$. Concatenăm în mod repetat şirurile $pref(p(k,j))$ (în ordine crescătoare a lui j), până când şirul obţinut conţine cel puţin $len(S)-1$ caractere (sau am considerat deja toate cele $NK(k)$ şiruri). $pref(k)$ constă din primele $len(S)-1$ caractere ale şirului obţinut (sau din întreg şirul, dacă acesta conţine mai puţin de $len(S)-1$ caractere). În mod similar, concatenăm (de la dreapta la stânga) şirurile $suf(p(k,j))$ (în ordine descrescătoare a lui j), până când şirul obţinut conţine cel puţin $len(S)-1$ caractere (sau am considerat deja toate cele $NK(k)$ şiruri). $suf(k)$ constă din ultimele $len(S)-1$ caractere ale şirului obţinut (sau din întreg şirul, dacă acesta conţine mai puţin de $len(S)-1$ caractere).

Vom proceda apoi după cum urmează. Iniţializăm un şir $X=suf(p(k,1))$. Apoi considerăm, pe rând, toate poziţiile $j=2, ..., NK(k)$. Să considerăm poziţia j la care am ajuns. Construim şirul X' prin concatenarea şirului X cu $pref(p(i,j))$. Căutăm de câte ori apare şirul S în şirul X' şi incrementăm $NA(k)$ cu acest număr de apariţii. Calculul numărului de apariţii se poate realiza prin forţă brută. După aceasta, concatenăm la sfârşitul şirului X şirul $suf(p(i,j))$. Dacă şirul X are acum cel puţin $len(S)$ caractere, atunci setăm X ca fiind egal cu subşirul format din ultimele $len(S)-1$ caractere ale sale.

Observăm că, la fiecare pas, şirul X are cel mult $len(S)-1$ caractere şi, prin urmare, şirul X' are cel mult $2 \cdot len(S)-2$ caractere.

Complexitatea acestei soluţii este $O(L \cdot (len(S))^2)$, unde L este suma valorilor $NK(i)$ ($1 \leq i \leq N$) sau, dacă folosim o metodă mai eficientă pentru a calcula numărul de apariţii al lui S (de ex., algoritmul KMP), complexitatea poate fi redusă la $O(L \cdot len(S))$.

Capitolul 6. Geometrie Computatională

Problema 6-1. Volumul reuniunii a N hiper-dreptunghiuri

Se dau N ($1 \le N \le 50$) hiper-dreptunghiuri d-dimensionale ($1 \le d \le 10$ şi $N^d \le 500.000$), cu laturile paralele cu axele de coordonate. Pentru fiecare hiper-dreptunghi i se dau coordonata minimă şi cea maximă în fiecare dimensiune j ($x_1(i,j)$ *şi* $x_2(i,j)$, $x_1(i,j) < x_2(i,j)$). Determinaţi volumul reuniunii celor N hiper-dreptunghiuri.

Soluţie: Vom sorta independent cele $2 \cdot N$ coordonate ale celor N hiper-dreptungiuri în fiecare dimensiune j ($1 \le j \le N$), eliminând duplicatele, astel încât $xs(j,1) < ... < xs(j,nx(j))$ ($xs(j)$=coordonatele sortate în dimensiunea j; $nx(j)$=numărul de coordonate distincte din dimensiunea j). În timpul sortării vom reţine pentru fiecare dreptunghi i indicii $ix_1(i,j)$ şi $ix_2(i,j)$ unde apar $x_1(i,j)$ şi $x_2(i,j)$ în $xs(j)$.

Vom calcula o matrice d-dimensională V, având $(nx(1)-1) \cdot ... \cdot (nx(d)-1) = O(N^d)$ celule. Elementul i din dimensiunea j a matricii V ($1 \le i \le nx(j)-1$) corespunde intervalului $[xs(j,i),xs(j,i+1)]$. Astfel, fiecare celulă $(c(1), ..., c(d))$ a lui V corespunde unui hiper-dreptunghi d-dimensional $[xs(1,c(1)),xs(1,c(1)+1)]x...x[xs(d,c(d)),xs(d,c(d)+1)]$ – vom defini volumul celulei $(c(1), ..., c(d))$, $cvol(c(1), ..., c(d))$ ca fiind volumul hiper-dreptunghiului d-dimensional asociat. Vom seta valoarea fiecărei celule din V la 0.

Considerăm apoi fiecare hiper-dreptunghi i dintre cele N şi parcurgem toate celulele $(c(1), ..., c(d))$, cu $ix_1(i,j) \le c(j) < ix_2(i,j)$; pentru fiecare astfel de celulă, incrementăm $V(c(1), ..., c(d))$ cu 1. La final, iniţializăm valoarea volumului reuniunii Vol la 0 şi apoi parcurgem toate celulele din matricea V. Dacă pentru o celulă $(c(1), ..., c(d))$ avem $V(c(1), ..., c(d)) > 0$, atunci adunăm la Vol valoarea $cvol(c(1), ..., c(d))$. Complexitatea acestei soluţii este $O(N^{d+1})$.

O soluţie mult mai simplă, dar cu dezavantajul de a fi aproximativă, este următoarea. Calculăm hiper-dreptunghiul minim *MBR* care include toate cele N hiper-dreptunghiuri date. *MBR* are coordonata minimă în dimensiunea j ($1 \le j \le d$) egală cu $min\{x_1(i,j) | 1 \le i \le N\}$ şi coordonata maximă în aceeaşi dimensiune egală cu $max\{x_2(i,j) | 1 \le i \le N\}$. Vom genera apoi, în mod aleator, un număr de M puncte aflate în interiorul lui *MBR*. Pentru fiecare punct, numărăm în câte dintre cele N hiper-dreptunghiuri este inclus. Dacă este inclus în cel puţin un hiper-dreptunghi, atunci vom incrementa un contor *nin* (iniţializat la 0). La final, o valoare aproximativă a volumului reuniunii este $nin/M \cdot Volum(MBR)$ (unde $Volum(MBR)$ reprezintă volumul lui *MBR*, care este uşor de calculat). Pentru a obţine o aproximare cât mai bună ar trebui să generăm un număr foarte mare de puncte (M foarte mare). Complexitatea acestei abordări este, însă, $O(M \cdot N)$.

Problema 6-2. Puncte maximale

Se dau N puncte în plan (punctul i e la coordonatele $(x(i), y(i))$). Oricare 2 puncte au coordonatele x şi y diferite. Un punct i se numeşte *maximal* dacă nu există un alt punct y, astfel încât $x(j) > x(i)$ şi $y(j) > y(i)$. Determinaţi mulţimea de puncte maximale.

Soluţie: Vom introduce punctele pe rând, în ordine crescătoare a indicelui lor, menţinând un arbore echilibrat cu punctele maximale de până acum (sortate după coordonata x). Pentru a nu trata cazuri particulare, vom introduce la început un punct $(-\infty,+-\infty)$ şi un punct $(+\infty,-\infty)$. Aceste două puncte vor fi mereu puncte maximale. Când introducem punctul $(x(i),y(i))$,

căutăm în arbore punctul j cu cea mai mică coordonată $x(j)$ mai mare decât $x(i)$. Dacă $y(j)>y(i)$, atunci punctul i nu este punct maximal şi putem să îl ignorăm. Dacă $y(j)<y(i)$, atunci vom iniţializa un indice k la predecesorul punctului j găsit. Cât timp $y(k)<y(i)$ executăm următoarele acţiuni:

(1) găsim k', predecesorul punctului k în arbore;

(2) ştergem punctul k din arbore (nu mai este punct maximal);

(3) $k=k'$.

La sfârşit, vom adăuga şi punctul i în arbore. Complexitatea acestui algoritm este $O(N \cdot log(N))$.

Problema 6-3. Puncte dominate

Se dau N puncte în plan (punctul i e la coordonatele $(x(i), y(i))$ şi are o pondere $w(i) \geq 0$). Pentru fiecare punct i dorim să determinăm suma ponderilor punctelor $j \neq i$ cu proprietatea $x(j) \leq x(i)$ şi $y(j) \leq y(i)$. Oricare *2* puncte se află amplasate la coordonate diferite.

Soluţie: O primă variantă constă din introducerea tuturor punctelor într-o structură de date numită *range tree*. Această structură permite calcularea sumei ponderilor punctelor dintr-un dreptunghi $[xa,xb]x[ya,yb]$ în timp $O(log^2(N))$ (sau $O(log(N))$, folosind nişte tehnici speciale). Pentru fiecare punct i, determinăm suma ponderilor punctelor care se află în dreptunghiul $[-\infty,x(i)]x[-\infty,y(i)]$ (şi scădem din ea $w(i)$, căci se va considera şi punctul i).

O a doua variantă constă în sortarea punctelor crescător după X (şi pentru X egal, crescător după Y). Vom parcurge punctele în această ordine şi vom menţine o structură de date peste coordonatele $y(*)$ distincte şi sortate ale celor N puncte (arbore de intervale sau împărţire în grupuri de câte $sqrt(N)$ poziţii). Avem două posibilităţi:

(1) pentru un punct i determinăm suma ponderilor punctelor din intervalul $[-\infty,y(i)]$ (range query), reprezentând valoarea dorită; apoi incrementăm cu $w(i)$ suma ponderilor punctelor aflate la coordonata $y(i)$ (point update);

(2) pentru un punct i determinăm suma valorilor cu care a fost incrementată poziţia $y(i)$ (point query); apoi incrementăm cu $w(i)$ numărul de puncte din intervalul $[y(i),+\infty]$ (range update).

Arborele de intervale oferă o complexitate $O(log(N))$ pentru ambele operaţii în ambele cazuri. Folosind împărţirea în grupe de câte $sqrt(N)$ poziţii, pentru cazul *(1)* avem $O(sqrt(N))$ pentru range query şi $O(1)$ pentru point update (incrementăm numărul de puncte asociate poziţiei $y(i)$, cât şi grupei din care face parte $y(i)$); pentru cazul *(2)* avem $O(1)$ pentru point query şi $O(sqrt(N))$ pentru range update.

O arhitectură generică de folosirea a arborilor de intervale şi a împărţirii în grupe de câte $sqrt(N)$ a fost prezentată în [Andreica-CTRQ2008] şi [Andreica-COMM2008].

Problema 6-4. Oypara (Lotul de Informatică, România, 2006)

Se dau N ($1 \leq N \leq 100.000$) segmente verticale: capatele segmentului i au coordonatele $(x(i),yjos(i))$ şi $(x(i),ysus(i))$. Determinaţi dacă există o dreaptă care să intersecteze toate cele N segmente. Dacă o astfel de dreaptă există, determinaţi ecuaţia acesteia.

Soluţie: Dacă există o dreaptă care intersectează toate cele N segmente, atunci aceasta poate fi translatată şi rotită astfel încât să atingă două dintre capetele segmentelor. Astfel, putem alege oricare două capete de segmente şi verifica în timp liniar dacă dreaptă determinată de

cele două capete intersectează toate cele N segmente. Acest algoritm are complexitatea $O(N^3)$ şi este ineficient pentru limitele problemei.

Un algoritm mai bun este următorul. Considerăm mulţimea A ca fiind formată din toate capetele sus ale segmentelor şi mulţimea B conţinând toate capetele jos ale segmentelor. Determinăm înfăşurătoarea convexă a punctelor din A (*CHA*) şi infăşurătoarea convexă a punctelor din B (*CHB*). Acum trebuie să determinăm dacă există o dreaptă care separă cele două poligoane (*CHA* şi *CHB* sunt în semiplane opuse ale dreptei). Pentru aceasta vom roti două drepte paralele în jurul lui *CHA* şi *CHB*. Pornim cu o dreaptă verticală *DA* care atinge *CHA* în cel mai din stânga punct, şi cu o dreaptă verticală *DB* care atinge *CHB* în cel mai din dreapta punct. Ambele drepte vor fi rotite în sens trigonometric pe conturul poligonului corespunzător.

Pentru fiecare dreaptă menţinem vârful de sprijin din cadrul lui *CHA* (*CHB*). Pe măsură ce rotim dreptele, avem o serie de evenimente în care acestea îşi schimbă punctul de sprijin. De exemplu, dacă dreapta *DA* se sprijinea pe punctul *PA* şi este rotită până când se suprapune cu muchia $(PA,PA+1)$, atunci noul ei punct de sprijin va fi $PA+1$. Pentru fiecare eveniment reţinem care este dreapta care îşi schimbă punctul de sprijin (*DA* sau *DB*), precum şi cu câte grade a fost rotită dreapta din poziţia iniţială pentru a avea loc evenimentul. Bineînţeles, evinementele vor fi sortate după numărul de grade asociat acestora. Dacă înainte sau după apariţia vreunui eveniment se întâmplă ca *PB* şi un punct din interiorul lui *CHA* (de ex., media aritmetică a coordonatelor vârfurilor acestuia) se află în semiplane opuse faţă de *DA*, atunci *DA* separă *CHA* şi *CHB* (la fel, şi *DB* separă *CHA* şi *CHB*).

Acest algoritm are complexitate $O(N \cdot log(N))$. Calcularea celor două înfăşurători convexe durează $O(N \cdot log(N))$. Există $O(N)$ evenimente, care pot fi sortate în timp $O(N \cdot log(N))$. Pentru fiecare eveniment, testarea dacă *PB* şî un punct din interiorul lui *CHA* sunt în semiplane opuse faţă de *DA* se realizează în timp $O(1)$ (se calculează semnul unor ecuaţii).

Problema 6-5. Puncte într-un poligon convex

Se dă un poligon convex având N ($1 \le N \le 100.000$) vârfuri; vârful i are coordonatele $(x(i),y(i))$. Se dau, de asemenea, M întrebări de forma: se află punctul (xp, yp) în interiorul poligonului ?

Soluţie: Vom alege un punct (xc,yc) aflat strict în interiorul poligonului. De exemplu, xc (yc) ar putea fi media aritmetică a coordonatelor x (y) ale vârfurilor. Vom trasa segmentele (xc,yc)-$(x(i),y(i))$ de la punctul (xc,yc) la fiecare vârf al poligonului. Vom sorta apoi aceste segmente după panta pe care o formează dreapta lor suport cu axa *OX*. Pentru fiecare punct (xp,yp), vom calcula panta Pp a dreptei ce conţine punctele (xc,yc) şi (xp,yp), apoi vom căuta binar segmentul (xc,yc)-$(x(i),y(i))$ având cea mai mare pantă mai mică sau egală cu Pp (dacă nu există niciun astfel de segment, atunci punctul se află în intervalul de unghiuri dintre ultimul segment şi primul; astfel, vom considera $i=N$). Vom verifica apoi dacă (xp,yp) se află în interiorul triunghiului determinat de punctele (xc,yc), $(x(i),y(i))$ şi $(x(i+1),y(i+1))$ (considerând punctele de pe poligon în aceeaşi ordine în care apar în sortarea după pantă şi considerând că vârful $N+1$ este acelaşi cu vârful 1). Verificarea se poate face în felul următor: trebuie ca (xp,yp) să se afle de aceeaşi parte a lui $(x(i),y(i))$-$(x(i+1),y(i+1))$ ca şi punctul (xc,yc). Complexitatea acestui algoritm este $O(N \cdot log(N))$.

Un alt algoritm având aceeaşi complexitate se bazează pe următoarea idee. Se alege o dreaptă ce trece prin fiecare punct testat (de exemplu, o dreaptă orizontală). Se determină apoi, în timp $O(log(N))$ (folosind un algoritm corespunzător), dacă dreapta intersectează

poligonul convex şi, dacă da, se calculează punctele de intersecţie; se verifică apoi ca punctul testat să se afle pe segmentul determinat de cele două puncte de intersecţie.

Problema 6-6. Intersecţii între drepte şi un poligon convex

Se dă un poligon convex având *N* ($1 \leq N \leq 100.000$) vârfuri; vârful *i* are coordonatele *(x(i),y(i))*. Se dau, de asemenea, *M* întrebări de forma: Dându-se o dreaptă determinată de două puncte (x_1, y_1) şi (x_2, y_2), determinaţi dacă această dreaptă intersectează sau nu poligonul.

Soluţie: Vom alege un punct *(xc,yc)* aflat strict în interiorul poligonului. O primă variantă de rezolvare a problemei este următoarea. Vom transalata sistemul de coordonate în asa fel încât *(xc,yc)* să ajungă în originea sistemului. Vom dualiza dreptele suport ale laturilor poligonului. Fiecare dreaptă este dualizată într-un punct din planul dual. Vom determina înfăşurătoarea convexă *CH* a punctelor duale (în mod normal ar trebui ca pe înfăşurătoare să apară toate punctele, în ordinea în care apar dreptele suport pe conturul poligonului convex iniţial). Pentru fiecare dreaptă dată, vom calcula punctul dual corespunzător acesteia. Dreapta va intersecta poligonul iniţial doar dacă punctul dual nu se află în interiorul înfăşurătorii convexe duale *CH*. Pentru testarea rapidă (în *O(log(N))*) a incluziunii unui punct într-un poligon convex, putem folosi soluţia de la problema anterioară.

Problema 6-7. Ghirlandă (ACM ICPC NEERC, Rusia, 2000, enunţ modificat)

Se dă o ghirlandă ce constă din *N* ($1 \leq N \leq 10.000$) lămpi, aşezate una după alta, la diferite înălţimi. Înălţimile *H(i)* ($1 \leq i \leq N$) satisfac următoarele reguli:

- $H(1)=A$ ($10 \leq A \leq 1000$, dat)
- $H(i)=((x \cdot H(i-1)+y \cdot H(i+1))/p)-q$ ($2 \leq i \leq N-1$, $0<p, q, x, y \leq 10$)
- $H(i) \geq 0$ ($1 \leq i \leq N$)

Determinaţi cea mai mică valoare posibilă pentru *H(N)*.

Exemple:

N=8, *A=15*, *p=2*, *q=1*, *x=1*, *y=1* => *H(N)=9,75*
N=692, *A=532,81*, *p=2*, *q=1*, *x=1*, *y=1* => *H(N)=446113,34*

Soluţie: Prima soluţie constă în a căuta binar cea mai mică valoare posibilă pentru *H(2)*. Să presupunem că am ales o valoare candidată *HC* pentru *H(2)*. Cunoscând pe *H(i)* şi pe *H(i-1)* ($2 \leq i \leq N-1$), putem calcula pe *H(i+1)*. Avem $H(i+1)=p/y \cdot (H(i)+q)-x/y \cdot H(i-1)$.

Calculăm toate valorile *H(i)* şi verificăm ca toate să fie mai mari sau egale cu *0*. Dacă această condiţie este îndeplinită, atunci *HC* este o valoare fezabilă şi vom testa o valoare mai mică data viitoare; altfel, vom testa o valoare mai mare. *H(N)* este valoarea calculată corespunzătoate celei mai mici înălţimi fezabile *H(2)*.

Vom termina căutarea binară atunci cân intervalul de căutare are o lungime mai mică sau egală cu o constantă fixată în prealabil (de ex., 10^{-4}). Limita superioară *HMAX* iniţială este aleasă corespunzător (de ex., 10^6). Complexitatea acestei soluţii este $O(N \cdot log(HMAX))$.

Un algoritm cu complexitatea liniară (*O(N)*) este următorul. Vom scrie fiecare înălţime *H(i)* ($1 \leq i \leq N$) ca o funcţie liniară de forma $a(i) \cdot H(2)+b(i)$. Pentru *i=1* avem *a(1)=0* şi *b(1)=A*, iar pentru *i=2* avem *a(i)=1* şi *b(i)=0*. Pentru $3 \leq i \leq N$, avem: $a(i)=p/y \cdot a(i-1)-x/y \cdot a(i-2)$ şi $b(i)=p/y \cdot b(i-1)+p \cdot q/y-x/y \cdot b(i-2)$. Din aceste ecuaţii vom calcula cea mai mică valoare fexabilă pentru *H(2)* şi cea mai mare valoare fezabilă.

Iniţializăm $H2min=0$ şi $H2max=+\infty$. Din $a(i)\cdot H(2)+b(i)\geq 0$, obţinem $a(i)\cdot H(2)\geq -b(i)$. Dacă $a(i)=0$ şi ($-b(i)>0$), atunci nu există soluţie. Dacă $a(i)>0$, setăm $H2min=max\{H2min, -b(i)/a(i)\}$; dacă $a(i)<0$, setăm $H2max=min\{H2max, -b(i)/a(i)\}$. La sfârşit, dacă $H2min>H2max$, atunci nu există nicio soluţie. Altfel, $H2min$ este cea mai mică valoare fezabilă pentru $H(2)$, iar $H2max$ este cea mai mare valoare fezabilă. Dacă $a(N)\geq 0$, setăm $H(2)=H2min$; altfel, setăm $H(2)=H2max$. Pornind de la valorile $H(1)$ şi $H(2)$ putem calcula iterativ valorile $H(3\leq i\leq N)$, obţinând, astfel, valoarea $H(N)$ minimă căutată.

Problema 6-8. Cerc cu K puncte (TIMUS, enunţ modificat)

Se dau N ($3\leq N\leq 50.000$) puncte în plan. Determinaţi un cerc (de orice rază) care să conţină pe circumferinţă sau în interiorul său exact K ($0\leq K\leq N$) puncte. Se ştie că oricare 3 puncte nu sunt coliniare şi oricare 4 puncte nu sunt conciclice (nu se află pe acelaşi cerc).

Soluţie: Pentru $K=0$ putem alege un cerc de rază 0 care nu are centrul într-unul din punctele date, iar pentru $K=1$ alegem un cerc de rază 0 cu centrul într-unul din punctele date.

Pentru $K\geq 2$ am putea încerca toate combinaţiile de 2 sau 3 puncte care determină un cerc, iar apoi am verifica dacă fiecare din celelalte puncte se află în interiorul cercului sau în afara acestuia. Complexitatea acestei soluţii este, însă, prea mare ($O(N^4)$). O soluţie de complexitate $O(N\cdot log(N))$ este următoarea. Vom alege două puncte A şi B care sunt consecutive pe înfăşurătoarea convexă a celor N puncte (determină o latură a acesteia). Pentru determinarea acestor puncte nu trebuie, neapărat, să calculăm înfăşurătoarea convexă. Putem alege, de exemplu, punctul A cu coordonata x minimă, iar punctul B este cel pentru care dreapta-suport a segmentului A-B are panta minimă dintre toate celelalte drepte A-i ($1\leq i\leq N$, $i\neq A$).

Cercul determinat va avea centrul pe mediatoarea segmentului AB (mediatoare=dreapta perpendiculară pe segmentul AB şi care trece prin mijlocul acestuia). Vom considera fiecare din celelalte N-2 puncte i şi vom calcula coordonatele centrului cercului determinat de punctele A, B şi i. Vom asocia fiecărui centru distanţa $d(i)$ faţă de mijlocul segmentului AB. Dacă centrul determinat este „înspre interiorul" înfăşurătorii convexe (adică în semiplanul dreptei AB ce conţine celelalte N-2 puncte, atunci distanţa va fi pozitivă); dacă se află „înspre exteriorul" înfăşurătorii convexe (în semiplanul opus celorlalte puncte faţă de dreapta AB), vom lua distanţa cu semnul -.

Vom sorta apoi centrele cercurilor după distanţa calculată. Avem $d(1)\leq d(2)\leq\ldots\leq d(N-2)$. Dacă avem $K=2$, putem alege centrul cercului la o distanţă mai mică decât d(1) (această distanţă poate fi oricât de aproape de $-\infty$). Dacă $K\geq 3$, atunci putem alege centrul cercului oriunde în intervalul $[d(K-2), d(K-1))$ (considerând $d(N-1)=+\infty$).

Problema 6-9. Puncte în triunghiuri (Lotul Naţional de Informatică, 2003)

Se dau N ($3\leq N\leq 1.000$) puncte în plan, oricare trei necoliniare şi oricare două având coordonate X şi Y diferite. Se dau şi M ($0\leq M\leq 500.000$) întrebări de tipul: determinaţi câte puncte sunt cuprinse strict în interiorul triunghiului cu vârfurile în punctele A, B şi C ($1\leq A,B,C\leq N$). Găsiţi o modalitate eficientă de a răspunde la întrebări.

Soluţie: Voi prezenta pentru început o soluţie ce răspunde în timp $O(log(N))$ la fiecare întrebare. Vom preprocesa setul de puncte în felul următor: considerăm fiecare punct x şi sortăm circular celelalte N-1 puncte y în jurul său, în funcţie de panta dreptei x-y. La fiecare întrebare vom considera că A şi C sunt punctele cu cordonata x minimă şi maximă, iar B este

localizat între ele. Dacă *B* este deasupra segmentului *A-C*, atunci vom calcula câte puncte *NA* se află în intervalul de unghiuri determinat de semidreptele *A-C* şi *A-B*, în sensul de la *C* spre *B* (vom efectua două căutări binare în ordinea circulară a punctelor din jurul punctului *A*). Apoi vom calcula câte puncte *NC* sunt în intervalul de unghiuri din jurul punctului *C*, determinat de porţiunea ce începe la punctul *C* din semidreapta *AC* şi semidreapta *CB* (în sens trigonometric). Apoi vom calcula căte puncte *NB* se află în jurul punctului *B*, în intervalul de unghiuri determinat de porţiunea semidreptei *AB* ce începe în punctul *B* şi porţiunea semidreptei *CB* ce începe în punctul *B* (în sens trigonometric). Răspunsul la întrebare este *NA-NC+NB*.

Dacă *B* se află sub segmentul *AC*, atunci calculăm: *NA*=numărul de puncte din jurul lui *A*, aflate în intervalul de unghiuri determinat de semidreptele *AB* şi *AC* (în sens trigonometric) ; *NB*=numărul de puncte din jurul lui *B*, aflate în intervalul de unghiuri determinat de porţiunea semidreptei *AB* ce începe în punctul *B* şi semidreapta *BC* (în sens trigonometric) ; *NC*=numărul de puncte din jurul lui *C*, aflate în intervalul de unghiuri determinat de porţiunea semidreptei *AC* ce începe în punctul *C* şi porţiunea semidreptei *BC* ce începe în punctul *C* (în sens trigonometric). Rezultatul este *NA-NB+NC*.

Răspunsul la fiecare întrebare se poate da şi în timp *O(1)*. Vom precalcula numărul de puncte *NP(i,j)* aflate sub fiecare segment determinat de perechea de puncte *(i,j)* (unde *i* are coordonata *x* mai mică decât *j*). La o întrebare, dacă *B* se află între *A* şi *C* şi deasupra segmentului *A-C*, răspunsul este *NC(A,B)+NC(B,C)-NC(A,C)*. Dacă *B* se află sub segmentul *A-C*, răspunsul este *NC(A,C)-NC(A,B)-NC(B,C)*.

Pentru a calcula eficient (în mai puţin de $O(N^3)$) valorile dorite, vom proceda în felul următor. Vom sorta punctele în ordine crescătoare a coordonatei *x*: *p(1), ..., p(N)*. Vom parcurge descrescător, cu un indice *i=N-1,...,1* aceste puncte. Vom sorta circular punctele *p(i+1), ..., p(N)* în jurul punctului *i*. Să considerăm aceste puncte *c(i,1), ..., c(i,N-i)*, în ordine crescătoare a unghiului format de dreapta *p(i)-c(i,j)* cu axa *OX* ($1 \le j \le N-i$). Avem *NC(p(i), c(i,1))=0*. Pentru $2 \le j \le N-i$, avem: dacă *x(c(i,j))>x(c(i,j-1))*, atunci *NC(p(i), c(i,j))=NC(p(i), c(i,j-1))+NC(c(i,j-1), c(i,j))+1*; altfel, *NC(p(i), c(i,j))=NC(p(i), c(i,j-1))-NC(c(i,j), c(i,j-1))*.

Astfel, preprocesarea se poate realiza în timp $O(N^2 \cdot log(N))$. Cu puţină atenţie, ea se poate realiza chiar în timp $O(N^2)$, dacă nu sortăm de la capăt toate punctele în jurul fiecărui punct *p(i)* [Eppstein1992].

Problema 6-10. Ciclu hamiltonian fără intersecţii (TIMUS)

Se dau *N* ($3 \le N \le 1.000$) puncte în plan, oricare trei necoliniare. Determinaţi un poligon care are ca vârfuri cele *N* puncte şi ale cărui laturi nu se intersectează (cu excepţia faptului că oricare *2* laturi consecutive de pe poligon se „ating” în vârful comun).

Soluţie: O soluţie de complexitate $O(N^2)$ este următoarea. Se determină înfăşurătoarea convexă a celor *N* puncte. Vom nota această înfăşurătoarea cu *L(1)*. Eliminăm punctele din *L(1)* şi apoi determinăm înfăşurătoarea convexă a punctelor rămase. Notăm această mulţime cu *L(2)*. Din punctele rămase eliminăm punctele din *L(2)* ş.a.m.d. Practic, calculăm în mod repetat înfăşurători convexe ale punctelor rămase după eliminarea punctelor de pe înfăşurătorile convexe deja calculate.

Să presupunem că am obţinut *k* „straturi” de înfăşurători convexe: *L(1), ..., L(k)*. *L(k)* poate să constea şi dintr-un singur punct sau din două puncte. Vom determina poligonul *P* dorit după cum urmează. Iniţializăm *P(1)* cu poligonul convex *L(1)*. Apoi, pentru fiecare *i=2,...,k-1*, vom „uni” *P(i-1)* cu *L(i)*.

Vom căută o latură (a,b) a lui $P(i-1)$ pe care să o eliminăm din $P(i-1)$. De asemenea, vom căuta o latură (p,q) a lui $L(i)$ pe care să o eliminăm. Dacă putem trasa segmentele (a,p) şi (b,q) (sau (a,q) şi (b,p)), atunci trasăm aceste segmente şi unim $P(i-1)$ cu $L(i)$ (notând poligonul obţinut cu $P(i)$).

La sfârşit, dacă $|L(k)|\geq 3$, procedăm similar ca şi în cazurile $2\leq i\leq k-1$. Dacă $L(k)=\{x\}$, vom căuta o latură (a,b) a lui $P(k-1)$ pe care să o eliminăm şi pe care să o înlocuim cu segmentele (a,x) şi (b,x). Dacă $L(k)=\{x,y\}$, atunci vom căuta o latură (a,b) a lui $P(k-1)$ pe care să o înlocuim cu segmentele (a,x) şi (y,b) (sau cu (a,y) şi (x,b)), la care adăugăm segmentul (x,y).

Dacă implementăm algoritmul descris în mod direct, complexitatea sa este $O(N^3)$ (la fiecare pas i considerăm orice latură (a,b) din $P(i-1)$ cu orice latură (p,q) din $L(i)$ şi verificăm dacă se pot elimina, în timp $O(N)$). Complexitatea se poate reduce la $O(N^2)$ observând că, la fiecare pas i, putem alege orice latură (a,b) a lui $P(i-1)$ care aparţine şi lui $L(i-1)$. Astfel, la fiecare pas, latura (a,b) este fixată mai întâi şi variem apoi doar latura (p,q) din $L(i)$). De menţionat că toate înfăşurătorile convexe pot fi calculate în timp $O(N^2)$ (fiind necesară doar o singură sortare a punctelor, la început).

O soluţie mult mai simplă, de complexitate $O(N\cdot log(N))$ este următoarea. Determinăm lanţul inferior al înfăşurătorii convexe a celor N puncte. Lanţul inferior conţine cel mai din stânga şi cel mai din dreapta punct (pe care le denumim punctele A şi B). Sortăm apoi crescător, după coordonata x (şi, pentru x egal, descrescător după coordonata y), punctele care nu se află pe lanţul inferior calculat. La ordonarea obţinută adăugăm punctul A la început şi punctul B la sfârşit. Fie această ordonare extinsă $p(1)(=A), p(2), ..., p(k)(=B)$. Vom trasa segmentele $(p(i), p(i+1))$ ($i=1,...,k-1$). Poligonul obţinut din lanţul inferior al înfăşurătorii convexe şi din segmentele trasate după aceea este poligonul dorit.

Problema 6-11. Cuplaj fără intersecţii (Lotul Naţional de Informatică, România, 2000; TIMUS)

Se dau N ($3\leq N\leq 1.000$) puncte în plan, oricare trei necoliniare. Determinaţi un număr maxim de segmente care unesc câte două puncte dintre cele N date, astfel încât oricare două segmente nu se intersectează (şi nici nu se ating într-un punct).

Soluţie: O soluţie de complexitate $O(N^2)$ este următoarea. Se determină înfăşurătoarea convexă a celor N puncte. Fie această înfăşurătoare $CH(1)$. Dacă ea conţine un număr P par de puncte, atunci alegem $P/2$ laturi ale lui $CH(1)$, în mod alternativ (o latură da, apoi următoarea nu). Dacă ea conţine un număr P impar de puncte, atunci eliminăm ultimul punct din $CH(1)$ (şi unim primul punct de penultimul), obţinând un polgon convex cu un număr Q par de vârfuri. Pe acest poligon procedăm ca şi în cazul anterior (alegând $Q/2$ laturi, oricare 2 neconsecutive pe conturul poligonului). Eliminăm punctele astfel „cuplate" şi repetăm procedeul pentru mulţimea de puncte încă necuplate. Ne oprim doar atunci când rămânem cu un singur punct sau cu 0 puncte. Astfel, se pot trasa ($N/2$) segmente (parte întreagă inferioară), oriare două neintersectându-se în niciun punct.

O soluţie mai simplă, de complexitate $O(N\cdot log(N))$ este următoarea. Sortăm punctele crescător după coordonata x (iar pentru puncte cu aceeaşi coordonată x, crescător după coordonata y). Fie această ordonare $p(1), p(2), ..., p(N)$. Vom trasa segmentele $(p(2\cdot i-1), p(2\cdot i))$ ($1\leq i\leq N/2$). Se observă uşor că aceste segmente nu se intersectează. Alternativ, putem alege un punct (xo,yo) situat la stânga tuturor celor N puncte date, după care vom sorta punctele i în funcţie de panta formată de semidreapta $(xo,yo)-(x(i),y(i))$ şi axa OX. Fie această ordonare $p(1), p(2), ..., p(N)$. Vom trasa segmentele în acelaşi mod ca şi în soluţia anterioară.

Problema 6-12. Cerc de rază minimă ce conţine puncte cu suma ponderilor cel puţin egală cu o valoare dată

Se dau N ($3 \le N \le 1.000$) puncte în plan. Fiecare punct i ($1 \le i \le N$) se află la coordonatele $(x(i),y(i))$ şi are o pondere $w(i) \ge 0$. Determinaţi un cerc de rază minimă astfel încât suma ponderilor punctelor din interiorul sau de pe conturul cercului să fie cel puţin egală cu X.

Soluţie: Vom căuta binar raza *Rmin* a cercului. Pentru fiecare rază R candidată, va trebui să determinăm suma maximă a ponderilor unei submulţimi de puncte ce poate fi inclusă într-un cerc de rază R. Dacă această valoare este mai mare sau egală cu X, vom căuta raze mai mici în căutarea binară; altfel vom căuta raze mai mari.

Putem considera că cercul are pe conturul său unul din cele N puncte (dacă nu, mutăm cercul până când unul din punctele din interior ajunge pe contur). Vom considera fiecare punct p drept punct pe conturul cercului şi vom calcula suma maximă a ponderilor punctelor ce pot fi incluse într-un cerc de rază R ce are punctul p pe contur. Vom considera toate punctele $i \neq p$ şi vom calcula pentru fiecare distanţa $dist(i,p)$ dintre punctele i şi p. Dacă $dist(i,p) > 2 \cdot R$, atunci vom ignora punctul i. Altfel, vom calcula două unghiuri $u_1(i)$ şi $u_2(i)$. $u_1(i)$ şi $u_2(i)$ sunt unghiurile faţă de axa OX ale segmentului (p,C), unde C este centrul unui cerc de rază R ce conţine punctele p şi i pe contur. Este clar că există 2 astfel de unghiuri. Dacă considerăm că centrul cercului de rază R se roteşte în jurul punctului în sens trigonometric, unul din cele 2 unghiuri corespunde cazului în care punctul i intră în cerc, iar celălalt unghi corespunde cazului când punctul i iese din cerc. Vom seta $u_1(i)$ ca fiind unghiul de intrare (unghiul „mai mic") şi $u_2(i)$ ca fiind unghiul de ieşire (unghiul „mai mare"). Fiecare punct i care nu a fost ignorat este acum echivalent cu 2 intervale disjuncte de unghiuri $[u_1(i),u_2(i)]$ şi $[u_1(i)+2 \cdot \pi, u_2(i)+2 \cdot \pi]$, care au fiecare o pondere $w(i)$.

Va trebui să determinăm o valoare u pentru care suma ponderilor intervalelor ce conţin valoarea u să fie maximă. Este uşor de observat că u trebuie să fie unul din capetele celor $O(N)$ intervale. Vom sorta cele $O(N)$ capete de intervale şi apoi le vom parcurge de la stânga la dreapta, menţinând o valoare *WS*, reprezentând suma ponderilor intervalelor intersectate în momentul curent (iniţial, $WS=0$). Când întâlnim un capăt stânga $u_1(i)$ (sau $u_1(i)+2 \cdot \pi$) corepunzător unui punct i, incrementăm *WS* cu $w(i)$; când întâlnim un capăt dreapta $u_2(i)$ (sau $u_2(i)+2 \cdot \pi$), decrementăm *WS* cu $w(i)$.

Valoarea maximă pe care o atinge *WS* pe parcursul algoritmului (după fiecare incrementare sau decrementare) este suma maximă a ponderilor punctelor conţinute într-un cerc de rază R ce are punctul p pe contur. Pentru a rezolva problema faptului că intervalele sunt circulare, vom considera că fiecărui punct i îi corespund, de fapt, 2 intervale disjuncte: $[u_1(i),u_2(i)]$ şi $[u_1(i)+2 \cdot \pi,u_2(i)+2 \cdot \pi]$.

Pentru un punct p, problema se rezolvă în timp $O(N \cdot log(N))$, algoritmul de calcul al sumei maxime a ponderilor incluse într-un cerc de rază dată având complexitatea totală de $O(N^2 \cdot log(N))$. Pentru rezolvarea problemei iniţiale se mai adaugă la complexitate un factor de $O(log(RMAX))$, corespunzător căutării binare a razei minime (*RMAX* este valoarea maximă a unei raze candidat posibile). Variante ale acestei probleme au fost propuse la conrcursul de programare Bursele Agora şi la Olimpiada de Informatică a Europei Centrale (2006).

Problema 6-13. H (Happy Coding 2007, infoarena)

Se dau N ($1 \leq N \leq 65535$) segmente verticale, numerotate de la 1 la N. Un segment K este caracterizat prin valorile $X(K)$, $Yjos(K)$ şi $Ysus(K)$. Coordonatele capătului de jos al segmentului K sunt $(X(K),Yjos(K))$, iar coordonatele capătului de sus sunt $(X(K),Ysus(K))$.

Vrem să alegem două dintre aceste segmente si să le „tăiem" în aşa fel încât ambele să aibă aceeaşi valoare pentru *Yjos*, respectiv pentru *Ysus*. Presupunând că segmentele alese sunt A şi B, noua valoare a lui *Yjos* pentru fiecare din cele două segmente va fi $YJ=max\{Yjos(A),Yjos(B)\}$, iar noua valoare a lui *Ysus* pentru fiecare din cele două segmente va fi $YS=min\{Ysus(A),Ysus(B)\}$. Dacă nu este adevarată relaţia $YJ \leq YS$, atunci perechea de segmente (A,B) nu este validă. Pentru o pereche validă de segmente (A,B), după operaţia de „tăiere" a segmentelor, se vor uni segmentele printr-un segment orizontal, pentru a forma o figură care seamănă foarte mult cu litera *„H"*. Segmentul orizontal va fi trasat intre coordonatele $(X(A),Y)$ si $(X(B),Y)$, cu $YJ \leq Y \leq YS$ (nu este importantă valoarea exactă a coordonatei Y a segmentului trasat). După obţinerea literei *„H"*, se calculează „lungimea" acesteia. Lungimea literei *„H"* este definită ca fiind suma celor 3 laturi ale literei: $2\cdot(YS-YJ)+|XB-XA|$.

Determinaţi o pereche validă de segmente pentru care lungimea literei *„H"* obţinuite să fie maximă.

Exemplu:
N=2
X(1)=0, Yjos(1)=0, Ysus(1)=10
X(2)=5, Yjos(2)=5, Ysus(2)=20
Lungimea maximă a unei litere „H" este 15.

Soluţie: O primă soluţie, de complexitate $O(N\cdot log^2(N))$, este următoarea. Pentru fiecare segment vertical (X_1,Yj_1,Ys_1), vom dori să determinăm un segment vertical din stânga sa (X_2,Yj_2,Ys_2), împreună cu care formează o literă H de lungime maximă, în fiecare din următoarele 4 cazuri (ce depind de pozitia celui de-al doilea segment fata de primul):

- $Yj_1 \leq Ys_2 \leq Ys_1$ şi $Yj_2 \leq Yj_1$: în acest caz, litera H va avea $YJ=Yj_1$ şi $YS=Ys_2$
- $Yj_2 \leq Yj_1 \leq Ys_1 \leq Ys_2$: în acest caz, litera H va avea $YJ=Yj_1$ şi $YS=Ys_1$
- $Yj_1 \leq Yj_2 \leq Ys_2 \leq Ys_1$: în acest caz, litera H va avea $YJ=Yj_2$ şi $YS=Ys_2$
- $Yj_1 \leq Yj_2 \leq Ys_1$ şi $Ys_1 \leq Ys_2$: în acest caz, litera H va avea $YJ=Yj_2$ şi $YS=Ys_1$

Constatăm că, o dată ce am fixat unul din cele 4 cazuri, este foarte clară contribuţia fiecăruia din cele 2 segmente la lungimea literei H:

- cazul 1: contribuţia segmentului 1 este $L_1=X_1-2\cdot Yj_1$, iar cea a segmentului 2 este $L_2=-X_2+2\cdot Ys_2$
- cazul 2: contribuţia segmentului 1 este $L_1=X_1+2\cdot(Ys_1 - Yj_1)$, iar cea a segmentului 2 este $L_2=-X_2$
- cazul 3: contribuţia segmentului 1 este $L_1=X_1$, iar cea a segmentului 2 este $L_2=-X_2+2\cdot(Ys_2-Yj_2)$
- cazul 4: contribuţia segmentului 1 este $L_1=X_1+2\cdot Ys_1$, iar cea a segmentului 2 este $L_2=-X_2-2\cdot Yj_2$

În aceste condiţii, pentru fiecare caz, putem privi fiecare segment ca un punct într-un alt sistem de coordonate 2D, în care coordonata X corespunde lui *Yjos*, iar coordonata Y corespunde lui *Ysus*. În plus, fiecare punct are asociată o pondere W. Pentru fiecare caz construim cu toate cele N puncte un arbore de intervale *2D*, care utilizează $O(N\cdot log(N))$ memorie. Acest arbore de intervale este, de fapt, un arbore de intervale obişnuit pentru

coordonatele *X*, însă fiecare nod al arborelui menţine un vector cu coordonatele *Y* ale punctelor ce corespund intervalului nodului respectiv. În plus, pentru fiecare coordonată *Y* se menţine şi ponderea asociată punctului.

Pentru fiecare caz *C* şi fiecare segment vertical *S*, vom dori să găsim segmentul vertical din stânga sa cu care poate forma o litera *H* de lungime maximă. Acest segment corespunde unui punct ce se află într-un dreptunghi ce corespunde constrângerilor pentru coordonatele *Yjos* şi *Ysus* corespunzătoare cazului *C* şi care are ponderea *W* maximă. Pentru fiecare caz, interogările pot fi astfel puse încât dreptunghiul *[a,b]x[c,d]* în care căutăm punctul să aibă ori coordonată *c* egală cu $-\infty$, ori coordonata *d* egală cu $+\infty$. În aceste condiţii, pentru fiecare nod din arborele de intervale în care se ajunge prin „spargerea" segmentului *[a,b]*, va trebui să determinăm ponderea maximă a unui punct care are coordonata *Y* printre primele *z* sau printre ultimele *z* (unde *z* este determinat folosind căutare binară în fiecare nod al arborelui de intervale al coordonatelor *X* în care ajungem). Din acest motiv, putem să menţinem pentru fiecare nod un vector *wmax[z]* cu ponderea maximă a unui punct dintre primele *z* puncte ce corespund intervalului nodului, precum şi un vector similar ce corespunde ultimelor *z* puncte ale intervalului; dacă unul dintre capetele *c* sau *d* nu era egal cu $+/-\infty$, atunci trebuia să folosim *RMQ* (Range Minimum Query) pentru fiecare nod din arbore.

Observăm că, în modul în care am pus interogările, nu am specificat nicăieri că punctul cu pondere maximă trebuie să corespundă unui segment aflat în stânga segmentului pentru care facem interogarea. Totuşi, dacă punctul cu pondere maximă obţinut nu corespunde unui segment aflat în stânga segmentului curent, atunci vom obţine în mod sigur o literă *H* de lungime mai mare în momentul în care vom efectua interogarea pentru segmentul corespunzător punctului obţinut (acest lucru se datorează modului în care sunt definite ponderile). Mai trebuie să avem grijă ca, atunci când realizăm interogarea, punctul de pondere maximă să nu corespundă chiar segmentului pentru care facem interogarea (ceea ce se poate întâmpla, întrucât nu am impus nicio condiţie pentru a evita această situaţie). Pentru a evita acest lucru, o posibilitate este să folosim nişte limite ale dreptunghiului de interogare care să excluda punctele ce corespund unor segmente ce au exact acelaşi *Yjos* şi acelaşi *Ysus* ca şi segmentul curent. În felul acesta, însă, ajungem să nu luăm în considerare *H*-uri formate din *2* segmente care au acelaşi *Yjos* şi acelasi *Ysus* (dar coordonate *X* diferite). Acest caz trebuie tratat separat, printr-o simplă sortare (întâi după *Yjos*, apoi după *Ysus* şi apoi după *X*) şi parcurgere a segmentelor (segmentele având acelaşi *Yjos* şi *Ysus* aflându-se unul după altul în ordinea sortată). Complexitatea acestei solutii este $O(N \cdot log^2(N))$ şi foloseşte memorie $O(N \cdot log(N))$.

Soluţia prezentată poate fi optimizată la complexitate $O(N \cdot log(N))$, folosind o tehnică standard. Orice query corespunzător unui arbore de intervale *2D* poate fi redus de la o complexitate $O(log^2(N))$ la o complexitate $O(log(N))$. Practic, se va efectua o căutare binară a coordonatei *Y* inferioare şi superioare numai în cadrul rădăcinii arborelui de intervale. Apoi, din construcţia arborelui, vom memora pentru fiecare punct *P* al fiecărui nod, *4* pointeri:

(1) un pointer către punctul cu cea mai mică coordonată *Y* mai mare sau egală cu cea a punctului *P* şi care se află în intervalul de coordonate *X* al fiului stânga ;

(2) un pointer către punctul cu cea mai mică coordonată *Y* mai mare sau egală cu cea a punctului *P* şi care se află în intervalul de coordonate *X* al fiului dreapta ;

(3) un pointer către punctul cu cea mai mare coordonată *Y* mai mică sau egală cu cea a punctului *P* şi care se află în intervalul de coordonate *X* al fiului stânga ;

(4) un pointer către punctul cu cea mai mare coordonată *Y* mai mică sau egală cu cea a punctului *P* şi care se află în intervalul de coordonate *X* al fiului dreapta.

O parte din aceşti pointeri pot fi calculaţi în momentul în care, în faza de construcţie a arborelui de intervale *2D*, se interclasează coordonatele *Y* corespunzătoare fiului stânga şi fiului dreapta. Cealaltă parte a pointer-ilor se calculează realizând încă *2* parcurgeri ale coordonatelor *Y* sortate, una de la stânga la dreapta şi alta de la dreapta la stânga (în condiţiile în care am memorat pentru fiecare coordonată *Y* de la care din cei doi fii provine).

O alternativă la folosirea arborelui de intervale *2D* este folosirea unei structuri de date *2D* numită kd-tree, care poate oferi aceleaşi operaţii ca şi un arbore de intervale *2D*. Diferenţa constă în cantitatea de memorie folosită ($O(N)$, în loc de $O(N \cdot log(N))$), dar si în complexitatea căutarii în interiorul unui dreptunghi ($O(sqrt(N))$, în loc de $O(log^2(N))$ sau $O(log(N))$ cu optimizarea menţionată). Acest arbore memorează, pentru fiecare nod al său *x*, ponderea maximă a unui punct din subarborele lui *x*.

Problema 6-14. Regiuni (infoarena)

Se dau *N* hiper-plane şi *M* puncte într-un spaţiu *d*-dimensional ($1 \le N,M \le 1.000$). Un hiperplan are $d-1$ dimensiuni; de ex., pentru $d=3$, un hiperplan este un plan în spaţiu, iar pentru $d=2$, un hiperplan este o dreaptă. Niciun punct nu se află pe vreun hiperplan. Hiperplanele împart planul în regiuni. Spunem că *2* puncte sunt în aceeaşi regiune dacă nu există vreun hiperplan care să le despartă. Se cere să afişaţi numărul de grupuri de puncte, fiecare grup conţinând toate punctele din aceeaşi regiune.

Soluţie: Vom rezolva problema incremental, adăugând pe rând câte un hiperplan şi menţinând informaţia despre grupuri. Când adăugăm un hiperplan iterăm peste toate grupurile. Un grup va fi împărţit în alte două: punctele din stânga hiperplanului şi punctele din dreapta lui; e posibil ca unul din aceste două grupuri să fie vid, caz în care nu îl mai reţinem. Un grup *i* îl împărţim în două în $O(x(i))$ operaţii, unde $x(i)$ este numărul de elemente din grup. Toate grupurile vor avea în total $x(1)+x(2)+...+x(i)+...=M$ elemente, deci pentru a actualiza grupurile adăugând un hiperplan efectuăm $O(M)$ operaţii. Astfel, soluţia are complexitatea finală $O(N \cdot M)$.

Altă soluţie ar consta în determinarea pentru fiecare punct *i* a unui vector $v(i)$ cu *N* elemente, unde $v(i,j)=+1$ sau -1 (în funcţie de semispaţiul în care se află punctul *i* faţă de hiperplanul *j*). Dacă doi astfel de vectori asociaţi la două puncte sunt egali, atunci cele două puncte sunt situate în aceeaşi regiune. Pentru determinarea egalităţii vectorilor s-ar putea folosi o sortare naivă, obţinând o complexitate de $O(M \cdot N \cdot log(M))$, sau radix sort, pentru o complexitate de $O(M \cdot N)$.

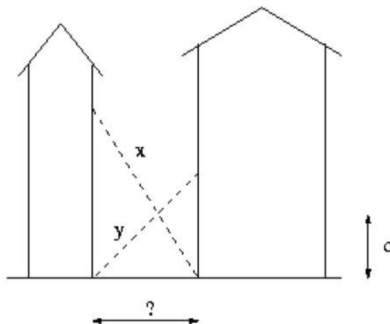
Altă variantă constă în introducerea acestor vectori într-un trie. Numărul de noduri terminale (frunze) ale trie-ului va reprezenta numărul de regiuni. Aceste soluţii folosesc $O(M \cdot N)$ memorie. O altă soluţie ar fi să obţinem un cod hash pentru fiecare vector; această soluţie are complexitatea $O(M \cdot N)$ ca timp şi $O(M+N)$ memorie. Pentru a face ca probabilitatea de coliziune să fie cât mai mică putem folosi câte $K \ge 2$ coduri diferite pentru fiecare vector.

Altă observaţie ar fi că vectorii sunt binari şi păstrând informaţia pe biţi folosim mai puţină memorie şi avem mult mai puţine operaţii la comparare dacă vrem să folosim soluţia în $O(M \cdot N \cdot log(M))$.

Problema 6-15. Scări încrucişate (UVA)

De o parte şi de alta a unei străzi se află *2* clădiri foarte înalte. De aceste *2* clădiri se află rezemate două scări. Prima scară are lungime *x* şi are baza la nivelul străzii, atingând clădirea

din dreapta, şi vârful este rezemat de clădirea din stânga. A doua scară are înălţimea *y* şi are baza la nivelul străzii, atingând clădirea din stânga, iar vârful este rezemat de clădirea din dreapta. Intersecţia celor *2* scări este la înălţimea *c*. Aflaţi lăţimea străzii.



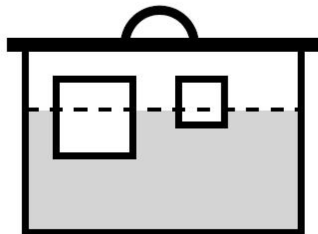
Exemplu: *x=30, y=40, c=10 => lăţimea străzii este 26.033.*

Soluţie: Vom căuta binar lăţimea *d* a străzii. Se observă că pe măsură ce lăţimea străzii creşte, înălţimea punctului de intersecţie scade; şi invers, pe măsură ce lăţimea străzii scade, înălţimea punctului de intersecţie creşte. Astfel, pentru o valoare fixată *d*, vom calcula *h(d)*=înălţimea punctului de intersecţie, dacă strada are lăţimea *d*. Dacă *h(d)<c*, atunci *d* este prea mare şi vom testa în continuare o valoare mai mică. Dacă *h(d)>c*, atunci d este prea mic şi vom testa în continuare o valoare mai mare.

Pentru a calcula *h(d)* vom proceda după cum urmează. Vom calcula *lx* şi *ly*, înălţimile (verticale) ale celor două scări: $lx=sqrt(x^2-d^2)$ şi $ly=sqrt(y^2-d^2)$. Fie *P* proiecţia punctului de intersecţie pe stradă şi fie *ld* distanţa de la capătul din stânga al străzii până la *P*. Din regulile de asemănare a triunghiurilor, avem: *h(d)/ly=ld/d* şi *h(d)/lx=(d-ld)/d=1-ld/d*. Înlocuind, obţinem *h(d)/lx=1-h(d)/ly => h(d)= lx·ly/(lx+ly)*.

Problema 6-16. Butoi cu apă (Olimpiada Baltică de Informatică 2003)

Avem un butoi cu apă de forma unui paraleliliped cu înălţime *HP* şi aria bazei *SP*. În acest butoi se toarnă, iniţial, apă până la înălţimea *HQ*. Apoi se introduc în apă *N* ($0 \le N \le 100.000$) paralelipipede. Paralelipipedul *i* are înălţimea *H(i)* ($0.01 \le H(i) \le HP$), aria bazei *S(i)* şi densitate *D(i)*. Densitatea apei este *DA*. Cele *N* paralelipipede nu se ating, iar suma ariilor bazelor lor nu depăşeşte *SP*. După aceasta, în partea de sus a butoiului se pune un capac, care împinge toate paralelipipedele care aveau partea de sus peste înălţimea *H* în interiorul butoiului. Determinaţi nivelul apei din butoi după punerea capacului. Dacă apa se revarsă în afara butoiului, menţionaţi acest lucru.



Soluţie: Dacă un paralelipiped *i* are densitatea *D(i)*, înseamnă că, în starea de echilibru, *HA(i)·S(i)·DA=(H(i)-HA(i))·S(i)·D(i)*, unde *HA(i)* este înalţimea paralelipipedului care se află

în interiorul apei. Astfel, obţinem $HA(i)\cdot(DA+D(i))=H(i)\cdot D(i)$ => $HA(i)=H(i)\cdot D(i)/(DA+D(i))$. Dacă nivelul curent HQ al apei este $HQ\geq HA(i)$ şi $HQ\leq HP-(H(i)-HA(i))$, atunci paralelipedul i este în echilibru. Dacă nivelul curent HQ al apei este $HQ<HA(i)$, vom spune că paralelipipedul i se află în starea -1; dacă $HA(i)\leq HQ\leq HP-H(i)+HA(i)$ vom spune că se află în starea 0; dacă $HQ>HP-H(i)+HA(i)$, vom spune că se află în starea 1.

Vom considera că paralelipipedele se introduc pe rând (de la paralelipipedul 1 la N). După introducerea fiecărui paralelipiped vom calcula noul nivel al apei din butoi. Vom menţine un min-heap cu evenimente: un eveniment este de forma *(înalţime apă, paralelipiped, stare)* şi se declanşează atunci când nivelul apei ajunge sau depăşeşte înălţimea specificată în cadrul evenimentului. Vom menţine, de asemenea, o valoarea SA ce reprezintă suma ariilor bazelor paralelipipedelor care se află în stările -1 sau 1. Iniţial $SA=0$ şi heap-ul este gol.

Să presupunem că am ajuns la paralelipipedul i ($1\leq i\leq N$) şi nivelul curent al apei este HQ. Mai întâi calculăm $HA(i)$. Dacă $HQ<HA(i)$, vom introduce în heap evenimentul $(HA(i), i, -1)$ şi vom incrementa SA cu $S(i)$; apoi vom seta o variabilă $Vol=VA(i)=HQ\cdot S(i)$. Dacă $HA(i)\leq HQ\leq HP-H(i)+HA(i)$, atunci vom adăuga în heap evenimentul $(HP-H(i)+HA(i), i, 0)$ şi vom seta $Vol=VA(i)=HA(i)\cdot S(i)$. Dacă $HQ>HP-H(i)+HA(i)$, vom incrementa SA cu $S(i)$ şi vom seta $Vol=VA(i)=(HQ-(HP-H(i)+HA(i))+HA(i))\cdot S(i)=(HQ-HP+H(i))\cdot S(i)$. $VA(i)$ reprezintă volumul din paralelipedul i care se află în apă. Volumul din butoi ocupat de apă şi părţi ale paralelipedelor aflate în apă este $HQ\cdot SP$. Vol reprezintă cu cât trebuie să crească acest volum, deoarece a crescut volumul părţilor cuburilor care se află scufundate în apă. Aşadar, în continuare, cât timp $Vol>0$, vom efectua următorii paşi:

(1) Vom calcula $HQ'=HQ+Vol/(SP-SA)$ (dacă $SP=SA$, setăm $HQ'=+\infty$).

(2) Dacă heap-ul conţine cel puţin un eveniment, atunci fie $E=(HE, j, s)$ evenimentul cu înălţimea minimă (evenimentele sunt ordonate în heap după această înălţime).

(2.1) Dacă $HE\leq HQ'$ atunci setăm $HQ'=HE$. HQ' va fi noul nivel al apei din butoi.

(2.2) Dacă $HQ'>HP$, atunci apa se revarsă din butoi şi întrerupem algoritmul. Altfel, vom decrementa Vol cu $(HQ'-HQ)\cdot(SP-SA)$, iar apoi vom seta $HQ=HQ'$.

(3) Cât timp heap-ul nu este gol şi cel mai mic eveniment din heap este $E=(HE=HQ, j, s)$, vom extrage evenimentul E din heap şi vom efectua următoarele acţiuni:

(3.1) dacă $s=-1$ atunci decrementăm SA cu $S(j)$ şi introducem în heap evenimentul $(HP-H(j)+HA(j), j, 0)$;

(3.2) dacă $s=0$ atunci incrementăm SA cu $S(j)$.

(4) După aceste acţiuni revenim la începutul ciclului care verifică dacă $Vol>0$.

Nivelul final al apei din butoi va fi HQ. Observăm că algoritmul are complexitatea $O(N\cdot log(N))$, deoarece se parcurg N paralelipipede şi se tratează cel mult $2\cdot N$ evenimente din heap (în complexitate $O(log(N))$ pentru fiecare).

Problema 6-17. Puncte (Olimpiada Internaţională de Informatică 2006)

Se dau $N+4$ puncte în plan ($0\leq N\leq 50.000$). Fiecare punct i ($1\leq i\leq N+4$) are o culoare $C(i)$, care poate fi *roşu* sau *verde*. Patru dintre puncte (numerotate cu $N+1$, ..., $N+4$) sunt colţurile unui dreptunghi: $(0,0)$, $(0,YMAX)$, $(XMAX,YMAX)$ şi $(XMAX,0)$. Primele 2 puncte ($(0,0)$ şi $(0,YMAX)$) au culoarea roşie, iar celelalte două au culoarea verde.

Dorim să determinăm un arbore parţial de acoperire $T(R)$ al punctelor roşii şi un arbore parţial de acoperire al punctelor verzi $T(V)$, astfel încât segmentele care formează cei 2 arbori

să nu se intersecteze. Dacă 2 puncte de aceeaşi culoare u şi v sunt adiacente în arborel parţial corespunzător, atunci se trasează segmentul u-v.

Se ştie că oricare 3 puncte dintre cele $N+4$ sunt necolineare.

Soluţie: Vom începe prin a împărţi dreptunghiul format din ultimele 4 puncte în două triunghiuri, prin trasarea unei diagonale. Fiecare dintre aceste 2 triunghiuri conţine 2 vârfuri de aceeaşi culoare şi un vârf de cealaltă culoare. Fie $S(a,b,c)$ mulţimea punctelor incluse în triunghiul cu vârfurile în punctele a, b şi c. Vom aplica următorul algoritm fiecăruia din cele 2 triunghiuri (pentru care determinăm, în timp $O(N)$, mulţimile $S(N+1, N+2, N+4)$ şi $S(N+2, N+3, N+4)$).

Să considerăm că tratăm triunghiul cu vârfurile în punctele a, b şi c, pentru care am calculat deja $S(a,b,c)$. Să presupunem, de asemenea, că vârfurile a şi b au aceeaşi culoare P, iar vârful c are cealaltă culoare, Q. Vom adăuga muchia a-b la arborele parţial de acoperire $T(P)$ (dacă nu a fost adăugată deja). Dacă toate punctele din $S(a,b,c)$ au culoarea P, atunci vom uni fiecare astfel de punct u cu unul din punctele a sau b (adăugând muchia u-a sau u-b la $T(P)$). Dacă există cel puţin un punct v de culoarea Q în triunghiul (a,b,c), atunci vom uni acest punct cu punctul c (adăugând muchia v-c la arborele parţial de acoperire $T(Q)$). Apoi vom construi triunghiurile (a,b,v), (a,v,c) şi (b,v,c). Fiecare astfel de triunghi are 2 vârfuri de aceeaşi culoare (a şi b; v şi c; v şi c) şi celălalt de culoare diferită. Vom determina $S(a,b,v)$, $S(a,v,c)$ şi $S(b,v,c)$ dintre punctele din ($S(a,b,c)\setminus\{v\}$) (într-un timp proporţional cu $O(|S(a,b,c)|)$). Apoi vom aplica recursiv algoritmul pentru fiecare cele trei triunghiuri construite: (a,b,v), (a,v,c) şi (b,v,c).

Când algoritmul se termină avem cei doi arbori parţiali $T(R)$ şi $T(V)$ gata construiţi, iar muchiile lor nu se intersectează. Algoritmul are complexitatea $O(N^2)$ în cel mai rău caz. Dacă am alege vârful v din interiorul unui triunghi (a,b,c) astfel încât cardinalele mulţimilor $S(a,b,v)$, $S(a,v,c)$ şi $S(b,v,c)$ să fie echilibrate, atunci am ajunge la o complexitate de ordinul $O(N\cdot log(N))$. În practică, putem folosi doar nişte euristici. Putem alege punctul v aleator dintre toate punctele v' din triunghiul (a,b,c) care au culoarea Q. Sau, dintre aceste puncte, îl putem alege pe cel care este cel mai aproape de: *(1)* centrul de greutate al triunghiului (care are drept coordonate mediile aritmetice ale coordonatelor x şi, respectiv, y, ale punctelor a, b şi c) ; sau *(2)* intersecţia mediatoarelor celor 3 laturi ale triunghiului ; sau *(3)* intersecţia medianelor celor 3 laturi ale triunghiului; sau *(4)* intersecţia înălţimilor celor 3 laturi ale triunghiului.

Problema 6-18. A K-a diagonală a unui poligon convex

Se dă un poligon convex cu N ($4\le N\le 100.000$) vârfuri. Considerând toate diagonalele sale sortate după lungime, determinaţi lungimea celei de-a K-a diagonale ($1\le K\le N\cdot(N-3)/2$).

Soluţie: Vom determina pentru fiecare vârf i ($0\le i\le N-1$) al poligonului vârful $Far(i)$ care se află la cea mai mare distanţă faţă de i. Pentru aceasta, avem 2 posibilităţi. Observăm că, dacă am parcurge vârfurile poligonului începând de la vărful următor lui i şi am merge circular până la vârful dinaintea lui i, lungimea segmentelor de la i la fiecare vârf creşte, după care scade. Vom considera funcţia $dist(i, j)$=distanţa dintre vârfurile numerotate cu i şi j. Pentru fiecare vârf i, definim funcţia $d_i(j)=dist(i, (i+j) \text{ mod } N)$ ($j=1,...,N-1$). Funcţia d_i este unimodală, adică este crescătoare (descrescătoare) până la un maxim (minim), după care este descrescătoare (crescătoare). Pentru a găsi punctul ei de maxim, putem folosi căutare binară pe „derivata" funcţiei. Dacă $d_i(j)-d_i(j-1)\ge 0$, atunci considerăm că suntem încă în partea

crescătoare a funcţiei; dacă $d_i(j)-d_i(j-1)<0$, atunci ne aflăm în partea descrescătoare a ei; în felul acesta, putem determina cel mai mare indice *jmax*, pentru care $d_i(jmax)-d_i(jmax-1)\geq 0$ ($1\leq jmax\leq N-1$; considerăm $d_i(0)=0$). Vom seta $Far(i)=(i+jmax) \bmod N$.

O altă modalitate de a determina indicele *jmax* este folosirea unei căutări ternare. Să presupunem că, în cadrul căutării, am decis că *jmax* se află în intervalul $[a,b]$ (iniţial, $[a,b]=[1,N-1]$). Vom calcula valorile $d_i((a+b)/3)$ şi $d_i(2\cdot(a+b)/3)$. Dacă $d_i(2\cdot(a+b)/3)\geq d_i((a+b)/3)$, atunci vom păstra, în continuare, intervalul $[a',b']=[(a+b)/3,\ b]$. Dacă $d_i(2\cdot(a+b)/3)<d_i((a+b)/3)$, atunci vom păstra, în continuare, intervalul $[a'',b'']=[a,\ 2\cdot(a+b)/3]$.

Putem calcula valorile $Far(i)$ şi în timp total $O(N)$. Vom considera valoarea virtuală $Far(-1)=0$. Vom considera apoi toate vârfurile $0\leq i\leq N-1$ (în ordine crescătoare). Pentru a calcula $Far(i)$, vom porni cu un indice $z=Far(i-1)$ şi, atâta timp cât cât $dist(i,z)-dist(i,\ (z+1) \bmod N)\leq 0$ şi ($z\neq(i-1+N) \bmod N$), îl vom incrementa pe z cu 1 (modulo N). Atunci când ieşim din ciclu, setăm $Far(i)=z$. Observăm că indicele z este incrementat, per total (pentru toate vârfurile), de $O(N)$ ori.

Dacă $K=N\cdot(N-1)/2$, atunci lungimea diagonalei căutate este $max\{dist(i,\ Far(i))\}$. Pentru alte valori ale lui K, vom căuta binar lungimea dorită. Fie L lungimea selectată în cadrul căutării binare şi fie LK lungimea dorită (necunoscută). Vom calcula numărul $Q(L)$ de diagonale care au lungimea mai mică sau egală cu L. Dacă $Q(L)\geq K$, atunci $L\geq LK$; dacă $Q(L)<K$, atunci $L<LK$. Vom determina, astfel, cea mai mică lungime L' pentru care $Q(L')\geq K$. Pentru a calcula $Q(L)$, vom considera, pe rând, fiecare vârf i ($0\leq i\leq N-1$) şi vom calcula $ND(i,L)$=numărul de diagonale care au un capăt în i şi lungimea mai mică sau egală cu L. Pentru aceasta, vom căuta binar cel mai mare indice $jmax_1$, între 0 şi $((Far(i)-i+N) \bmod N)$ pentru care $d_i(jmax_1)\leq L$. Apoi vom căuta binar cel mai mare indice $jmax_2$ între 0 şi $((i-Far(i)+N) \bmod N)$ pentru care $d_i(N-jmax_2)\leq L$ (considerăm $d_i(N)=0$). Vom seta apoi $ND(i,L)=jmax_1+jmax_2$. $Q(L)$ este egal cu $(ND(0,L)+...+ND(i,L)+...+ND(N-1,L))/2$ (deoarece fiecare diagonală i-j cu lungime mai mică sau egală cu L, este numărată atât în $ND(i,L)$, cât şi în $ND(j,L)$).

Complexitatea întregului algoritm este dominată de partea de căutare binară a lungimii celei de-a K-a diagonale, fiind de ordinul $O(N\cdot log(N)\cdot log(LMAX))$ (unde *LMAX* este lungimea maximă posibilă a unei diagonale). Întrucât lungimile diagonalelor sunt numere reale, căutarea binară a lungimii se va termina când intervalul de căutare are o lungime mai mică decât o constantă foarte mică, stabilită în prealabil.

Problema 6-19. Numărarea pătratelor

Se dau N ($4\leq N\leq 3.000$) puncte (distincte) în plan. Punctul i ($1\leq i\leq N$) are coordonatele $(x(i),\ y(i))$ Determinaţi câte pătrate cu laturile paralele şi cu vârfurile în 4 din cele N puncte date există.

Soluţie: Vom considera fiecare pereche de puncte (i,j) şi vom verifica dacă formează colţurile opuse ale unui pătrat. Dacă $|x(i)-x(j)|=|y(i)-y(j)|$, atunci există şansa ca ele să fie colţurile opuse ale unui pătrat. Mai trebuie doar să verificăm dacă există şi celelalte 2 puncte (celalte 2 colţuri opuse) printre cele N puncte date. Cel mai uşor ar fi să verificăm dacă punctele $(min\{x(i),\ x(j)\},\ min\{y(i),\ y(j)\})$, $(min\{x(i),\ x(j)\},\ max\{y(i),\ y(j)\})$, $(max\{x(i),\ x(j)\},\ min\{y(i),\ y(j)\})$, $(max\{x(i),\ x(j)\},\ max\{y(i),\ y(j)\})$ există printre cele N. Dacă toate aceste 4 puncte există, atunci incrementăm un contor NP cu 1 (iniţial, NP=0). Valoarea finală a lui NP este numărul de pătrate căutat.

Pentru a verifica dacă un punct *(a,b)* există în mulţimea noastră de puncte, putem sorta toate cele *N* puncte (crescător după *x* şi, în caz de egalitate, crescător după *y*). Apoi, putem căuta binar punctul *(a,b)* în vectorul de puncte sortate. O soluţie de complexitate mai bună ar fi să folosim un hash, în care introducem toate punctele *(x(i), y(i))*. Apoi ar trebui doar să verificăm dacă o pereche *(a,b)* există în hash, verificare ce poate fi realizată în timp *O(1)*.

Complexitatea întregului algoritm este $O(N^2 \cdot log(N))$ (dacă folosim căutare binară) sau $O(N^2)$ (dacă folosim hash-ul).

Algoritmul poate fi extins pentru a calcula numărul de pătrate având orice orientare. Când se consideră o pereche de puncte *(i,j)*, ne gândim că acestea ar putea reprezenta una dintre diagonalele unui pătrat. În acest caz, calculăm coordonatele celorlalte două vârfuri ale pătratului şi verificăm dacă acestea se află printre cele *N* puncte date. Complexitatea soluţiei rămâne neschimbată.

Problema 6-20. Cutii (infoarena)

Se dau *N* ($1 \leq N \leq 3.500$) cutii *d*-dimensionale ($1 \leq d \leq 5$). Fiecare cutie *i* ($1 \leq i \leq N$) are dimensiunile *(l(i,1), …, l(i,d))* şi o pondere *w(i)*. Dimensiunile celor *N* cutii în fiecare dimensiune *j* reprezintă o permutare a numerelor *{1,...,N}*. Dorim să determinăm o secvenţă de cutii cu suma maximă a ponderilor, *a(1), ..., a(K)*, astfel încât cutia *a(i+1)* să poată fi introdusă în cutia *a(i)* ($1 \leq i \leq K-1$). Cutia *a(i+1)* poate fi introdusă în cutia *a(i)* dacă există o permutare *p* a dimensiunilor sale, astfel încât $l(a(i),j) > l(a(i+1),p(j))$ (pentru fiecare $1 \leq j \leq d$).

Soluţie: Vom genera un şir *S* ce conţine $M=N \cdot d!$ cutii. Fiecare cutie este introdusă în şir de *d!* ori, câte o dată pentru fiecare permutare a dimensiunilor sale (dimensiunile asociate fiecărei apariţii a ei fiind permutate corespunzător). Fiecare apariţie a unei cutii *i* în *S* are ponderea *w(i)*. Problema cere acum determinarea unei secvenţe *a(i), ..., a(K)*, de sumă maximă a ponderilor, astfel încât $l(a(i),j) > l(a(i+1),j)$ ($1 \leq i \leq K-1$; $1 \leq j \leq d$). Observăm că deşi o cutie *i* apare de *d!* ori, nicio apariţie a cutiei *i* nu va putea fi introdusă într-o altă apariţie a aceleiaşi cutii.

Vom sorta cutiile după prima dimensiune, astfel încât să avem $l(S(1),1) \geq \ldots \geq l(S(M),1)$. De asemenea, vom construi un arbore multi-dimensional (range tree sau kd-tree) peste toate cele *M* cutii, considerând doar dimensiunile *2,...,d*. Fiecărei (apariţii a unei) cutii *i* ($1 \leq i \leq M$) i se asociază un punct *p(i)=(l(i,2), ..., l(i,d))*, care are iniţial ponderea $wp(i)=-\infty$. Cu ajutorul arborelui vom putea determina în mod eficient cea mai mare pondere a unui punct din cadrul unui dreptunghi multidimensional *R* (în timp $O(log^{d-1}(M))$ pentru range tree, sau $O(M^{1-1/d})$ pentru kd-tree). Fiecare nod al arborelui va menţine ponderea maximă asociată unui punct din subarborele său şi are asociat un interval multidimensional. Când se ajunge la un nod din arbore care este conţinut în *R*, se întoarce valoarea memorată în acel nod.

Vom parcurge apoi cutiile în ordinea sortată. Când ajungem la o cutie *i*, vom determina ponderea maximă a unui punct din arbore care are coordonatele *(x(2), ..., x(d))* în intervalul $l(i,j) < x(j) \leq N$ ($1 \leq j \leq d$). Fie această pondere *wmax* (dacă nu se găseşte niciun punct în intervalul multidimensional, se întoarce $wmax=-\infty$). Setăm $wp(i)=w(i)+max\{0, wmax\}$ şi modificăm valoarea *wp(i)* asociată punctului *i* din arbore: pornim de la punct (care se află într-o frunză a arborelui) şi mergem în sus, actualizând valorile menţinute în fiecare nod al arborelui (valoarea maximă dintre toate punctele din subarbore) ; cum ponderile asociate punctelor pot doar să crească, actualizarea unei valori *v* se face setând $v=max\{v, wp(i)\}$. Răspunsul la problemă este $max\{wp(i) | 1 \leq i \leq M\}$.

Problema 6-21. Grăsanul 1

Se dă un coridor *2D* de lungime infinită şi înălţime *H*. În interiorul acestui coridor se găsesc *N* puncte: punctul *i* la coordonatele *(x(i), y(i))*. Un „grăsan” (având forma unui cerc) se află în interiorul coridorului la coordonata $x=-\infty$ şi vrea să ajungă la coordonata $x=+\infty$. Determinaţi raza maximă pe care o poate avea „grăsanul”, astfel încât acesta să se poată deplasa din poziţia iniţială în poziţia finală fără ca vreun punct să intre în interiorul acestuia.

Soluţie: Vom căuta binar raza *R* a „grăsanului”, în intervalul *[0,H/2]*. Pentru o rază *R* fixată, vom verifica dacă „grăsanul” poate trece printre puncte cu raza respectivă. Vom asocia fiecărui punct *i* un cerc de coordonate *(x(i), y(i))* şi rază *R*. Vom construi un graf ce va conţine *N+2* noduri, corespunzătoare celor *N* cercuri şi a pereţilor de sus şi de jos ai cordidorului. Între *2* cercuri avem muchie dacă acestea se intersectează. Peretele de sus este „mutat” cu *R* unităţi în jos, iar cel de jos este mutat cu *R* unităţi în sus. Avem muchie între un perete şi un cerc dacă cercul intersectează peretele (considerând poziţia „mutată” a peretului). Vom verifica apoi dacă nodurile corespunzătoare celor *2* pereţi sunt în aceeaşi componentă conexă a grafului construit. Dacă da, atunci raza *R* este prea mare şi va trebui să căutăm o rază mai mică; dacă nu, raza *R* este fezabilă şi vom testa în continuare raze mai mari. Complexitatea acestei soluţii este $O(N^2 \cdot log(RMAX))$, unde *RMAX* este lungimea intervalului în care are loc căutarea razei.

Un algoritm de complexitate mai bună este următorul. Construim graful complet ce conţine *N+2* noduri: cele *N* puncte şi cei *2* pereţi. Între oricare *2* puncte ducem o muchie având costul egal cu jumătatea distanţei dintre puncte. Între orice punct şi fiecare din cei *2* pereţi ducem o muchie egală cu jumătatea distanţei de la punct la perete (jumătate din lungimea perpendicularei de la punct la perete). Costul muchie dintre cei *2* pereţi este egal cu jumătatea distanţei dintre ei. Acum vom calcula un drum de la peretele de jos la cel de sus, în care costul maxim al unei muchii este minim posibil (adică este cel mai mic dintre toate drumurile). Acest drum se poate determina în timp $O(N^2)$ (modificând uşor algoritmul lui Dijkstra). Costul maxim al unei muchii de pe acest drum este egal chiar cu raza maximă a „grăsanului”.

Problema poate fi generalizată la cazul în care grăsanul nu este un cerc, ci un poligon convex oarecare, pentru care vrem să determinăm factorul de scalare maxim relativ la un punct din interiorul poligonului (de ex., centrul acestuia). De asemenea, coridorul poate consta din mai multe segmente, nu doar din *2* pereţi (de ex., din *2* linii poligonale). Putem căuta binar factorul maxim de scalare *F*. Construim apoi câte un poligon convex identic ca cel al grăsanului, scalat cu factorul *F*, în jurul fiecărui punct. Apoi construim acelaşi graf ca şi mai înainte: dacă poligoanele a *2* puncte se intersectează, ducem muchie între aceste *2* puncte. Segmentele ce formează cele *2* linii poligonale ce delimiteaxă coridorul trebuie şi ele „îngroşate” cu forma poligonului convex, folosind o tehnică ce se numeşte „suma Minkovski”.

Cel mai simplu caz este când robotul este un pătrat iar segmentele liniilor poligonale sunt ortogonale: segmentele se vor translata cu o distanţă corespunzăţoare factorului de scalare, în aşa fel încât să micşoreze „grosimea” coridorului; capetele segmentelor vor fi translatate pe diagonala pătratului, iar apoi vor fi unite de segmentele translatate (în aşa fel, unele segment pot creşte sau scădea în lungime). Apoi, dacă cel puţin un segment al lanţului poligonal *i* (*i=1,2*) intersectează poligonul corespunzător unui punct *j*, ducem muchie de la nodul corespunzător lanţului poligonal *i* la nodul corespunzător punctului *j*. Dacă există un drum în

graf între cele *2* lanţuri poligonale, atunci grăsanul nu poate trece de la un capăt la celălalt al coridorului, factorul de scalare *F* fiind prea mare.

Şi a doua metodă, bazată pe determinarea unui drum minim, poate fi generalizată în acest caz. Între *2* puncte *i* şi *j* ducem o muchie de cost *F*, unde *F* este factorul de scalare minim pentru care poligoanele construite în jurul punctelor *i* şi *j* s-ar intersecta. Apoi, să considerăm un lanţ poligonal *i* (*i=1,2*). Vom considera toate segmentele *q* ce compun lanţul poligonal şi fiecare punct *j*. Vom calcula *F(i,q,j)*=factorul de scalare minim pentru care poligonul construit în jurul punctului *j* ar intersecta segmentul *q* al lanţului *i*, dacă acesta ar fi „modificat” conform factorului de scalare. Costul muchiei între lanţul *i* şi punctul *j* este *min{F(i,q,j)}*.

Problema 6-22. Numărul de intersecţii ale unor segmente pe cerc

Se dă un cerc de rază *R*, cu centrul în origine. Se dau, de asemenea, *N* segmente, având ambele capete pe circumferinţa cercului. Determinaţi numărul de intersecţii între segmente.

Soluţie: Alegem *2* puncte diferite *A* şi *B* pe circumferinţa cercului şi împărţim circumferinţa în *2* părţi: partea de la *A* la *B*, şi partea de la *B* la *A*. Cele *N* segmente se împart în *3* categorii:

1) cele care au ambele capete în partea de la *A* la *B* ;

2) cele care au ambele capete în partea de la *B* la *A* ;

3) cele care un capăt în partea de la *A* la *B* şi celălalt capăt în partea de la *B* la *A*.

Segmentele din categoriile *1)* şi *2)* sunt tratate foarte simplu (fiecare caz în mod independent). Vom considera că „întindem” fiecare din cele *2* părţi, până când acestea ajung nişte segmente orizontale (de lungime egală cu partea respectivă din circumferinţă). După această „întindere”, fiecare segment a devenit un interval pe partea respesctivă (putem obţine acelaşi rezultat dacă asociem fiecărui capăt al unui segment o coordonată *x* egală cu distanţa de la începutul părţii din circumferinţă până la capătul respectiv; distanţa este calculată de-a lungul circumferinţei). Problema s-a redus acum la determinarea numărului de perechi de intervale care se intersectează.

Vom sorta capetele stânga şi dreapta ale intervalelor. Vom parcurge apoi aceste capete în ordinea sortată, menţinând un contor *nopen* şi un contor *nint* (ambele sunt iniţial *0*). La fiecare capăt stânga întâlnit incrementăm *nopen* cu *1*. Când întâlnim un capăt dreapta incrementăm *nint* cu *(nopen-1)*, după care decrementăm *nopen* cu *1*. Pentru segmentele din categoria *3* vom proceda după cum urmează. Vom sorta aceste segmente în funcţie de distanţa capătului lor din partea de la *A* la *B* faţă de punctul *A* (distanţa e măsurată pe circumferinţa cercului, în sensul de la *A* până la capătul segmentului). Vom sorta apoi aceste segmente după distanţa calculată şi le vom asocia numere de la *1* la *M* (*M*=numărul de segmente din categoria *3*). Vom calcula apoi distanţa capătului de pe partea de la *B* la *A*, tot faţă de punctul *A*, măsurată pe circumferinţa cercului, în sensul de la punctul *A* până la capătul respectiv. Vom sorta apoi segmentele după această a doua distanţă calculată. În acest fel am obţinut o permutare a numerelor de la *1* la *M* (considerând numerele asociate la prima sortare, în ordinea de la a doua sortare). Numărul de intersecţii între segmentele din categoria *3* este egal cu numărul de inversiuni ale acestei permutări. Aşadar, numărul total de intersecţii între *N* segmente având capetele pe circumferinţa unui cerc se poate calcula în timp *O(N·log(N))*.

Problema poate fi extinsă după cum urmează. Considerăm un poligon convex cu *S* laturi şi *N* segmente având capetele pe perimetrul poligonului. Dorim să determinăm numărul de intersecţii dinre segmente. Vom proceda la fel ca în cazul cercului, împărţind conturul

poligonului în 2 părţi, în funcţie de 2 puncte A şi B. Diferenţa constă în faptul că acum distanţele se vor calcula de-a lungul conturului poligonului, şi nu de-a lungul circumferinţei unui cerc. Pentru simplitate vom alege A şi B ca fiind 2 vârfuri consecutive ale poligonului (de ex., vărfurile $B=1$ şi $A=2$). Vom considera că partea de la A la B este cea ce conţine laturile 2-3, ..., $(N-1)$-N, N-1. Vom calcula distanţele $d(i)=d(i-1)$+lungimea laturii $(i-1)$-i, pentru fiecare vărf al poligonului ($d(2)=0$ şi $d(1)=d(N)$+lungimea laturii N-1). Pentru a calcula distanţa de la un punct de pe latura 1-2 la vârful A, calculăm direct distanţa Euclideană. Pentru a calcula distanţa de la un punct aflat pe latura i-$(i+1)$ (inclusiv latura N-1) la vârful A, calculăm distanţa euclideană la vârful i, la care adăugăm valoarea $d(i)$. Problema poate fi extinsă şi la alte tipuri de curbe închise şi convexe.

Problema 6-23. Triunghiuri de arie număr întreg

Se dau N ($1 \le N \le 100.000$) puncte în plan, aflate la coordonate întregi $(x(i), y(i))$ ($1 \le i \le N$). Determinaţi numărul de triunghiuri cu vârfurile în 3 puncte diferite dintre cele N puncte, a căror arie este un număr întreg.

Soluţie: Să presupunem că am ales 3 puncte distincte, a, b şi c. Fie $S(a,b,c)=|x(a)\cdot y(b)-x(b)\cdot y(a)+x(b)\cdot y(c)-x(c)\cdot y(b)+x(c)\cdot y(a)-x(a)\cdot y(c)|$. Aria triunghiului determinat de cele 3 puncte este: $S(a,b,c)/2$. Aşadar, aria triunghiului este un număr întreg dacă $S(a,b,c)$ este un număr par. Paritatea lui $S(a,b,c)$ depinde doar de parităţile coordonatelor celor 3 puncte. Aşadar, vom împărţi cele N puncte în 4 clase. Clasa (p,q) conţine acele puncte i pentru care $x(i) \bmod 2=p$ şi $y(i) \bmod 2=q$ ($p=0,1$; $q=0,1$). Vom calcula $np(p,q)$=numărul de puncte ce aparţin clasei (p,q). Vom considera apoi orice combinaţie de câte 3 clase: $(p(1),q(1))$, $(p(2),q(2))$, $(p(3),q(3))$ (vom considera clasele ordonate lexicografic întâi după $p(*)$, apoi după $q(*)$; cele 3 clase nu trebuie să fie neapărat distincte). Vom considera că al j-lea punct al triunghiului ($1 \le j \le 3$) aparţine clasei $(p(j),q(j))$. Vom calcula acum expresia $S(a,b,c)$, pentru cazul în care cele 3 puncte a, b şi c aparţin celor 3 clase considerate (în loc de $x(a)$, $y(a)$, $x(b)$, $y(b)$, $x(c)$, $y(c)$, vom folosi $p(1)$, $q(1)$, $p(2)$, $q(2)$, $p(3)$ şi $q(3)$). Dacă expresia calculată este pară, vom incrementa un contor $ntri$ (iniţial 0) cu numărul de triunghiuri de arie întreagă obţinute: dacă cele 3 clase sunt distincte ($p(j) \ne p(k)$ sau $q(j) \ne q(k)$ pentru orice $1 \le j<k \le 3$), atunci incrementăm $ntri$ cu $np(p(1),q(1))\cdot np(p(2),q(2))\cdot np(p(3),q(3))$; dacă 2 dintre clase sunt identice între ele şi a treia e diferită, atunci: fie j şi k indicii celor 2 clase identice şi $l=6-j-k$ indicele clasei diferite => vom incrementa $ntri$ cu $np(p(l),q(l))\cdot(np(p(j),q(j))\cdot(np(p(j),q(j))-1))/2$; dacă toate cele 3 clase sunt identice, atunci vom incrementa $ntri$ cu $np(p(1),q(1))\cdot(np(p(1),q(1))-1)\cdot(np(p(1),q(1))-2)/6$. Complexitatea algoritmului este $O(N)$.

Problema 6-24. Graful zonelor unui graf planar

Se dă un graf neorientat cu N ($1 \le N \le 1.000$) noduri, desenate ca puncte în plan, şi M ($N-1 \le M \le 3\cdot N-6$) muchii. Muchia dintre 2 noduri i şi j este desenată sub forma segmentului care uneşte punctele corespunzătoare celor 2 noduri. Fiecare muchie (i,j) are un cost $c(i,j)>0$. Segmentele corespunzătoare a oricare 2 muchii nu se intersectează, iar graful este conex. Se dau apoi NP puncte (x,y) şi dorim să determinăm pentru fiecare dintre ele costul agregat minim al segmentelor ce trebuie intersectate în cazul în care vrem să ajungem din punctul (x,y) dat undeva la infinit (de ex., la $(+\infty,+\infty)$). Agregarea costurilor se realizează folosind o funcţie de agregare comutativă şi crescătoare, agg (de ex., agg=*adunare, înmulţire, max*, etc.).

Soluţie: Vom sorta circular muchiile adiacente fiecărui nod *i*, în ordinea unghiurilor formate cu axa *OX*. Să presupunem că nodul *i* este adiacent cu *deg(i)* muchii. Sortarea după unghi a muchiilor în jurul nodului *i* determină *deg(i)* intervale de unghiuri (al *k*-lea interval de unghiuri este între a *k*-a şi a *(k+1)*-a muchie adiacentă cu nodul *i*, în ordinea sortată; considerăm că muchia *deg(i)+1* este muchia *1* în ordinea sortată, iar muchia *0* este muchia *deg(i)* în ordinea sortată).

Vom construi un graf nou *GZ*, neorientat, în care fiecare nod este o pereche *(i,k)* ce corespunde celui de-al *k*-lea interval de unghiuri al nodului *i* ($1 \le j \le deg(i)$). Să considerăm o muchie *(i,j)* şi să presupunem că ea se află pe poziţia *a* în ordinea sortării în jurul nodului *i* şi pe poziţia *b* în ordinea sortării în jurul nodului *j*. Vom introduce muchii de cost *0* în noul graf între perechile *(i,a)* şi *(j,b-1)*, precum şi între perechile *(i,a-1)* şi *(j,b)*. Vom introduce apoi muchii de cost *c(i,j)* între perechile *(i,a)* şi *(j,b)*, precum şi între perechile *(i,a-1)* şi *(j,b-1)*.

Vom alege cel mai din stânga punct *q* şi vom considera intervalul său de unghiuri ce conţine unghiul π; fie acesta intervalul cu numărul *p*. Vom calcula apoi costul agregat minim al unui drum de la perechea *(q,p)* la toate celelalte perechi din graful *GZ* (folosind muchiile lui *GZ*). Pentru fiecare pereche *(q',p')* se calculează, astfel, *cmin(q',p')*=costul agregat minim pentru a ajunge la perechea *(q,p)*=costul agregat minim pentru a ajunge în exteriorul grafului şi, deci, inclusiv la $(+\infty,+\infty)$. Pentru aceasta folosim orice algoritm de calcul al drumurilor de cost minim, în care *2* costuri *CA* şi *CB* nu se adună pentru a obţine costul unui drum mai lung *CC=CA+CB*, ci se agregă: *CC=agg(CA,CB)*. Toate perechile *(q'',p'')* la care s-a putut ajunge cu costul *cmin(q'',p'')=0* (inclusiv *(q,p)*, care are, prin definiţie, *cmin(q,p)=0*) au un drum către exterior fără să intersecteze niciun segment. Complexitatea acestui pas este $O((N+M) \cdot log(N+M))$.

Dacă valorile *cmin(i,j)* sunt limitate superior de o constantă *CMAX* mică, atunci putem folosi *CMAX+1* cozi de prioritate, coada *Qu(co)* corespunzând perechilor *(q',p')* care, până la momentul respectiv din cadrul rulării algoritmului, au *cmin(q',p')=co*; astfel, complexitatea pasului se reduce la *O(N+M+CMAX)*.

În continuare, pentru fiecare punct *(x,y)* dat, va trebui să determinăm o pereche *(i,k)* cu proprietatea că unghiul format de segmentul *S(i,x,y)* ce uneşte nodul *i* şi punctul *(x,y)* formează cu axa *OX* un unghi ce se află în al *k*-lea interval de unghiuri al nodului *i*, precum şi că segmentul *S(i,x,y)* nu intersectează vreo muchie a grafului. Varianta cea mai simplă constă în considerarea, pe rând, a fiecărui nod *i* al grafului. Calculăm unghiul *u(i,x,y)* format de *S(i,x,y)* cu axa *OX* şi căutăm binar (sau liniar) intervalul *k* de unghiuri al nodului *i* în care se găseşte unghiul *u(i,x,y)*. Apoi considerăm pe rând fiecare muchie a grafului şi verificăm dacă *S(i,x,y)* se intersectează cu muchia respectivă.

O metodă mai bună se bazează pe folosirea unor tehnici de *„point location"*. Apoi, după găsirea unei perechi *(i,k)* corespunzătoare, costul total minim al segmentelor ce trebuie intersectate de un drum ce pleacă din *(x,y)* şi ajunge la $(+\infty,+\infty)$ este *cmin(i,k)*.

Problema 6-25. Acoperire cu poligoane

Se dau *N* ($1 \le N \le 10.000$) poligoane (nu neapărat convexe) având cel mult $M \le 100.000$ de vârfuri în total. Se dă, de asemenea, un poligon „mare". Verificaţi dacă reuniunea celor *N* poligoane este egală cu poligonul „mare" şi, în acelaşi timp, suprafaţa comună a oricare *2* poligoane este *0*. Niciunul din poligoane (inclusiv cel „mare") nu au unghiuri interioare fix egale cu π (adică laturi consecutive aflate una în prelungirea celeilalte) sau cu $2 \cdot \pi$.

Soluţie: Pentru început, vom verifica că suma ariilor celor *N* poligoane este egală cu cea a poligonului „mare” (în caz contrar, răspunsul este negativ). Dacă răspunsul este afirmativ, vom considera toate vârfurile celor *N* poligoane (plus cele ale poligonului „„mare”) şi le vom asocia identificatori unici (de la *1* la *M'*=numărul total de vârfuri). Putem realiza acest lucru folosind o tabelă hash (iniţial vidă). Parcurgem toate vârfurile (în orice ordine) şi, dacă găsim o cheie în tabela hash egală cu coordonatele vârfului, atunci identificatorul vărfului va fi valoarea asociată cheii; altfel, incrementăm cu *1* un contor *cnt* (iniţial *0*) şi adăugăm în tabela hash perechea *(cheie=coordonatele vârfului, valoare=cnt)* (identificatorul vârfului va fi *cnt*).

Pentru fiecare poligon „mic” *p*, vom considera fiecare vârf *i* al său (*i* este identificatorul unic) şi vom calcula intervalul de unghiuri *[u(p,i), v(p,i)]* pe care le formează cele 2 laturi ale poligonului adiacente vârfului *i* cu axa *OX*. Vom asocia acest interval vârfului *i*. Vom sorta apoi (după capătul stânga), pentru fiecare vârf *i*, toate intervalele de unghiuri asociate acestuia. Intervalele asociate trebuie să nu se intersecteze (decât să se atingă la capete), iar suma „lungimilor” lor să fie $2\cdot\pi$ (acoperă tot planul în jurul vârfului *i*) sau π (acoperă un semiplan în jurul lui *i*; acest caz apare dacă *i* se află pe o latură a poligonului „mare” sau pe o latură a unui poligon *p*, al cărui vârf nu este) sau, dacă *i* este un vărf al poligonului „mare”, trebuie ca reuniunea intervalelor de unghiuri ale poligoanelor „mici” din jurul lui *i* să fie egală cu intervalul de unghiuri format de laturile poligonului „mare” în vărful *i* (iar suma lungimilor acestor intervale de unghiuri trebuie să fie egală cu unghiul format de poligonul „mare” în vărful *i*).

Mai trebuie să verificăm şi că fiecare latură a unui poligon mic se suprapune peste o latură a poligonului mare sau peste o altă latură a unui alt poligon mic. Pentru aceasta, vom sorta laturile în funcţie de unghiul format de dreapta lor suport cu axa *OX* (şi, în caz de egalitate, în funcţie de punctul de intersecţie al dreptei suport cu axa *OX* ; dacă dreapta suport este orizontală, vom folosi intersecţia cu *OY*). Toate laturile ce formează acelaşi unghi şi dreptele lor suport au acelaşi punct de intersecţie cu *OX* (sau, dacă sunt orizontale, cu *OY*) formează o grupă şi vor fi sortate după coordonata *x* minimă a lor (şi, în caz de egalitate, după coordonata *y* minimă a lor). În cadrul sortării vom considera şi laturile poligonului mare. Vom împărţi apoi fiecare grupă în subgrupe. Laturile dintr-o grupă pot fi privite ca nişte intervale. Vom calcula reuniunea intervalelor din fiecare grupă. Dacă această reuniune constă din mai multe intervale disjuncte, atunci vom împărţi grupa în mai multe subgrupe, fiecare corespunzând laturilor incluse într-unul din intervalele reuniunii (altfel, vom avea o singură subgrupă ce conţine întreaga grupă).

Trebuie să verificăm acum că laturile din aceeaşi sungrupă pot fi „colorate” în două culori, astfel încăt reuniunea laturilor de aceeaşi culoare să fie identică cu reuniunea laturilor de cealaltă culoare. Pentru aceasta, putem folosi soluţia problemei *2-13*, unde *K=2* – aplicând algoritmul de acolo, trebuie ca răspunsul să fie afirmativ şi, în plus, fiecare latură să facă parte dintr-un din cele *K* submulţimi.

Complexitatea întregului algoritm este $O(M'\cdot log(M')))$.

Problema 6-26. Frontiera unui poligon (ACM ICPC NEERC 2000)

Se dă un poligon convex cu *N* ($1\leq N\leq 200$) vârfuri; vârful *i* are coordonatele *(x(i), y(i))*. În interiorul poligonului se află *M* ($1\leq M\leq 10.000$) de puncte *speciale*. Dorim să eliminăm o parte dintre punctele de pe frontiera poligonului astfel încât poligonul convex obţinut din punctele rămase să conţină în interiorul său toate cele *M* puncte speciale şi: *(a)* să aibă perimetru minim ; sau *(b)* să aibă arie minimă.

Soluţie: Vom considera graful orientat complet *G* cu *N* noduri, unde fiecare nod *i* corespunde vârfului *i* al poligonului şi fiecare muchie orientată (*i->j*) corespunde segmentului orientat (*i->j*) (sunt $N \cdot (N-1)$ segmente în total).

Pentru segmentul (*1->2*) vom calcula semnul *S* (pozitiv sau negativ) al punctelor speciale faţă de segment. Pentru fiecare segment orientat (*i->j*) vom verifica că toate punctele speciale au acelaşi semn *S* faţă de segmentul (*i->j*) ca şi în cazul segmentului (*1->2*). Dacă nu au toate acelaşi semn egal cu *S*, asociem segmentului (*i->j*) costul $c(i,j)=+\infty$. Dacă au acelaşi semn, vom asocia următorul cost muchiei (*i->j*): în cazul *(a)*, *c(i,j)*=lungimea segmentului *(i,j)*; în cazul *(b)*, $c(i,j)=(x(j)-x(i)) \cdot ((y(i)-ymin)+(y(j)-ymin))/2$ (aria cu semn a trapezului ce are două laturi paralele cu axa *OY*, latura de sus este segmentul (*i->j*), iar latura de jos este proiecţia segmentului (*i->j*) pe dreapta orizontală *y=ymin*), unde *ymin*=coordonată *y* minimă a unui vârf al poligonului. Folosind algoritmul Floyd-Warshall (cunoscut şi ca algoritmul Roy-Floyd), vom calcula costul total minim al unui drum în *G* între oricare două vârfuri *i* şi *j* (*D(i,j)*). Vom considera iniţial că $D(i,i)=+\infty$. Astfel, după rularea algortmului, *D(i,i)* va fi egal cu costul celui mai scurt ciclu orientat ce conţine nodul *i*. Vom alege nodul *i* pentru care *D(i,i)* este minim. Ciclul corespunzător lui *D(i,i)* va determina noua frontieră a poligonului (vârfurile ce rămân pe frontieră, celelalte fiind eliminate). Să observăm că, în cazul *(a)*, costul ciclului reprezintă perimetrul minim, iar în cazul *(b)*, reprezintă aria totală minimă.

Să observăm că putem considera şi problema mai generală, în care costul segmentului dintre o pereche de puncte *(i,j)* este dat ca fiind *c(i,j)* (şi nu trebuie să fie neapărat lungimea segmentului sau costul modificat din cazul ariei). În acest caz, am dori ca costul total al laturilor poligonului rămas după eliminarea unor vârfuri (şi care conţine în interior toate punctele speciale) să fie cât mai mic.

Dacă am dori să minimizăm costul maxim al unui segment *(i,j)* ce face parte din soluţie, atunci putem căuta binar acest cost maxim *Cmax*. Să presupunem că în căutare binară am fixat un cost maxim *C'*. Vom păstra doar segmentele *i->j* unde $c(i,j) \leq C'$, apoi vom verifica dacă în graful orientat obţinut există vreun ciclu. O variantă pentru verificarea acestui lucru constă în calcularea componentelor tare conexe ale grafului. Dacă există cel puţin o componentă tare conexă care conţine două vârfuri, atunci graful conţine un ciclu (orientat). O altă soluţie constă în utilizarea algoritmului de detecţie a ciclurilor prezentat în soluţia problemei *3-1*. O variantă mult mai simplă în acest caz este să ignorăm sensurile segmentelor, să determinăm componentele conexe ale grafului şi să verificăm că există cel puţin o componentă conexă care nu este arbore.

O versiune mai complexă a problemei este următoarea: Dorim ca poligonul convex obţinut să aibă exact *K* laturi, iar costul agregat al laturilor sale să fie minim (unde putem folosi o funcţie de agregare *aggf*, precum suma, maxim, etc.). În acest caz, vom utiliza un algoritm de programare dinamică. Vom calcula *Cmin(i,j,p)*=costul total minim pentru a selecta *p* segmente din intervalul de vârfuri *i, i+1, …, j* (circular), capătul primului segment să fie în vârful *i*, iar capătul celui de-al *p*-lea segment să fie în *j*. Vom calcula aceste valori în ordine crescătoare a lungimilor intervalelor (circulare) *i…j*. Avem *Cmin(i,j,0)=0*, *Cmin(i,j,1)=c(i,j)* şi $Cmin(i,j,p)=min\{aggf(Cmin(i,q,p-1), c(q,j)) \mid q$ se află în intervalul circular *i…j* şi $q \neq j\}$. Răspunsul este $min\{aggf(Cmin(i,j,K-1), c(j,i))\}$.

Dacă aplicăm direct algoritmul, obţinem o complexitate $O(N^3 \cdot K)$. Pentru anumite funcţii de agregare şi modalităţi de definire a costurilor *c(*,*)* ale segmentelor, putem să utilizăm structuri de date eficiente pentru determinarea lui *Cmin(i,j,p)*, reducând astfel complexitatea (eventual, chiar până la $O(N^2 \cdot K)$).

O soluţie mai ineficientă (de complexitate $O(N^3 \cdot K^2)$ se poate obţine utilizând următoarea recurenţă: $Cmin(i,j,p \geq 2)=min\{aggf(Cmin(i,q,r),\ Cmin(q,j,p-r))\ |\ q$ se află în intervalul circular $i…j$, $q \neq j$, $1 \leq r \leq p-1\}$. O astfel de formulare ar fi necesară dacă costurile nu ar fi asociate segmentelor, ci, eventual, unor alte tipuri de structuri (de ex., triunghiuri).

Dacă limita K nu este impusă, atunci putem renunţa la al treilea termen al stării calculate de programrea dinamică (vom renunţa la parameterul p), reducând complexitatea soluţiei la $O(N^3)$ (sau mai puţin, eventual chiar până la $O(N^2)$, în funcţie de funcţia de agregare şi funcţiile de cost date). În acest caz, vom începe cu $Cmin(i,j)=c(i,j)$ şi vom încerca apoi să micşorăm această valoare: $Cmin(i,j)=min\{Cmin(i,j),\ min\{aggf(Cmin(i,q),\ c(q,j))\ |\ q$ este în intervalul $i…j$ şi $q \neq j\}$ (dacă utilizăm al doilea tip de recurenţă, atunci vom avea $Cmin(i,j)=min\{Cmin(i,j),\ min\{aggf(Cmin(i,q),\ Cmin(q,j))\ |\ q$ este în intervalul $i…j$ şi $q \neq j\}$).

Problema 6-27. Distanţe punct-poligon

Se dă un poligon convex fixat cu N vârfuri ($3 \leq N \leq 100.000$) şi M ($1 \leq M \leq 200.000$) întrebări de forma: determinaţi distanţa de la un punct dat (x,y) la poligon (distanţa este 0 dacă punctul este inclus în poligon).

Soluţie: Întâi trebuie să determinăm dacă punctul este în poligon sau nu (folosind una din tehnicile deja discutate la altă problemă). Apoi, dacă punctul se află în exterior, atunci trebuie să determinăm punctele de tangenţă *low* şi *high*. *low* (*high*) este cel mai de jos (sus) vârf al poligonului cu proprietatea că dreapta care uneşte punctul (x,y) de vârful *low* (*high*) are întreg poligonul convex de aceeaşi parte. Mai întâi vom găsi latura $(q,q+1)$ a poligonului, cu proprietatea că punctul se află în intervalul de unghiuri determinat de aceste două vârfuri (relativ la un punct central fixat din interiorul poligonului, (xc,yc)). Considerăm că vârful q este mai jos decât vârful $q+1$. Apoi vom efectua două căutări binare, pentru a determina vârfurile *low* şi *high*. Proprietatea pe care o folosim în căutarea binară este că pentru orice punct p de la $q+1$ la *high* (fără *high*) sau de la q la *low* (fără *low*), avem că punctele $p-1$ şi $p+1$ sunt de părţi opuse ale dreptei determinate de punctul p şi punctul (x,y) din întrebare. După determinarea vârfurilor *low* şi *high*, folosim proprietatea că funcţia de distanţă de la punctul dat în întrebare şi până la poligon este o funcţie uni-modală (cu un singur minim) între punctele *low* şi *high* de pe poligonul convex (de pe partea dinspre punct). În acest caz putem folosi căutare ternnară sau căutare pe baza „derivatei”.

Capitolul 7. Combinatorică şi Teoria Numerelor

Problema 7-1. Furnica (Happy Coding 2007, infoarena)

O furnicuţă se află într-un muşuroi de formă pătratică, alcătuit din încăperi care au la rândul lor formă pătratică. Într-o zi, furnicuţa pleacă din încăperea ei, şi începe să se plimbe prin muşuroi. În fiecare zi, ea va trece din încăperea în care se află într-o încăpere alăturată, prin care, eventual, a mai trecut într-o zi anterioară. O încăpere alăturată se defineşte ca fiind un pătrat cu care camera (pătratul) în care se află furnicuţa are o latură în comun. Furnicuţa nu rămâne două zile la rând în aceeaşi cameră. Poziţia iniţială a furnicuţei este ori în centrul muşuroiului, ori în colţul din stânga-sus al acestuia, ca în figură:

Dându-se numărul de zile *t* care au trecut de la începutul plimbării furnicuţei şi poziţia sa iniţială (centru – ‚C' sau stânga-sus – ‚S'), să se determine numărul minim de încăperi din muşuroi în care trebuie căutată furnicuţa, pentru a fi siguri că aceasta va fi găsită. Numărul de încăperi de pe laturile muşuroiului se consideră a fi mult mai mare decât numărul de zile în care se plimbă furnicuţa

Soluţie:

- dacă poziţia de start este *‚C'*, atunci răspunsul este $(t+1)^2$
- dacă poziţia de start este *‚S'*, atunci:
  - dacă t este par, atunci răspunsul este $((t/2)+1)^2$
  - dacă t este impar, răspunsul este $((t+1)\cdot(t+2)/2)-((t+1)/2)^2$

Aceste formule (sau altele echivalente) pot fi obţinute folosind următorul raţionament: furnica se poate afla în orice poziţie aflată la o distanţă care are aceeaşi paritate ca şi t, faţă de poziţia iniţială. Pentru pozitia de start *‚C'*, aceste poziţii formează niste „romburi", iar pentru poziţia de start *‚S'*, aceste poziţii formează nişte diagonale „secundare". Dacă furnicuţa ar putea rămâne în aceeaşi încăpere de la o zi la alta, observăm că este suficient să considerăm cazul în care aceasta rămâne în camera iniţială la sfârşitul primei zile. Astfel, răspunsul ar fi suma dintre răspunsurile pentru t şi $t-1$, considerând constrângerile din enunţ (furnica nu rămâne 2 zile la rând în aceeaşi cameră).

Problema 7-2. Mulţimi (Happy Coding 2007, infoarena)

Considerăm mulţimea $[n]=\{1,...,n\}$ a primelor n ($1\leq n\leq 100.000$) numere naturale nenule. Multimile $A_1, ..., A_m$ acoperă $[n]$ dacă şi numai dacă oricare ar fi $1\leq i\leq n$ există $1\leq j\leq m$ astfel încât A_j să conţină pe i. Mulţimile $A_1, ..., A_m$ separă pe $[n]$ dacă şi numai dacă oricare ar fi $1\leq k,p\leq n$ există $1\leq j\leq m$ astfel încât cardinalul intersecţiei dintre A_j şi $\{k,p\}$ să fie 1 (practic există cel puţin o mulţime în care nu se află ambele elemente simultan). Pentru n dat, să se găsească m minim astfel încât $A_1, ..., A_m$ să acopere şi să separe mulţimea $[n]$. De asemenea, să se afişeze m mulţimi $A_1, ..., A_m$ care verifică această proprietate.

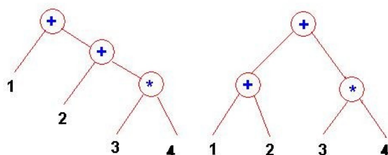
Soluţie: Se consideră matricea M cu n linii şi m coloane, unde $M[i][j]=1$, dacă i este în A_j, şi 0, altfel. Deoarece $A_1, ..., A_m$ separă multimea $[n]$, rezultă că oricare 2 linii ale matricei sunt diferite (dacă liniile k şi l ar fi egale, atunci k şi l nu pot fi separate). De aici rezultă că $n\leq 2^m$ (numărul de linii posibile, astfel încât oricare 2 să fie diferite). Aşadar, $m\geq log_2 n$. Vom arăta

că $m=(log_2 n)+1$. Presupunem prin absurd că $m=log_2 n$. Din principiul lui Dirichlet se obţine cu uşurinţă faptul că fie apare în matrice linia *00...0* (contradicţie cu condiţia de acoperire), fie matricea conţine *2* linii identice, ceea ce este absurd.

Pentru $m=(log_2 n)+1$ se poate da exemplul următor: A_j este mulţimea numerelor mai mici sau egale cu *n* care au *1* pe bitul *(j-1)* din reprezentarea lor binară pe *m* biţi (am numerotat biţii de la cel mai puţin semnificativ la cel mai semnificativ, începând cu bitul *0*).

Problema 7-3. Expresii Algebrice (Selecţie echipe ACM ICPC, UPB 2006)

O expresie algebrică poate fi reprezentată printr-un arbore. Putem evalua expresia parcurgând arborele ei corespunzător. Reprezentarea sub formă de arbore a unei expresii algebrice nu este obligatoriu unică. De exemplu, expresia *1+2+3*4* poate fi reprezentată prin următorii doi arbori:



La o privire atentă observăm că succesiunea operaţiilor + şi * se menţine, iar ordinea operanzilor rămâne neschimbată. În primul arbore ordinea evaluării va fi: *3*4 = 12; 2+12 = 14; 1+14 = 15*. În al doilea arbore, expresia va fi evaluată astfel: *1+2 = 3; 3*4 = 12; 3+12 = 15*. Ambele reprezentări produc rezultatul dorit. În această problemă vom considera expresii algebrice simple ce conţin doar numere dintr-o singură cifră, *'+'*, *'*'* şi paranteze. Expresia se evalueaza dupa regulile algebrice normale. Determinaţi numărul de reprezentări sub formă de arbore care evaluează expresia correct.

Exemple:

`1+2+3+4`	Numărul de arbori = **5**.
`(1+2)+(3+4)`	Numărul de arbori = **1**.
`1+2+3*4`	Numărul de arbori = **2**.
`1+2+(3*4)`	Numărul de arbori = **2**.
`1+*7`	Numărul de arbori = **0**.
`1+2*(3+(4*5)`	Numărul de arbori = **0**.

Soluţie: Vom calcula o matrie *T[i][j]* reprezentând numărul de arbori de evaluare a sub-expresiei dintre poziţiile *i* şi *j*. Dacă sub-expresia dintre aceste poziţii nu este validă (nu este parantezată corect, nu are o structură corespunzătoare de operanzi şi operatori, etc.), atunci *T[i][j]* va fi *0*. În caz contrar, dacă expresia dintre poziţiile *i* şi *j* este inclusă între paranteze, atunci *T[i][j]=T[i+1][j-1]*. Dacă nu este complet inclusă într-o pereche de paranteze, atunci există cel puţin un operator care nu este inclus în vreo paranteză. Se caută întâi operatorii *'+'* care nu sunt incluşi între paranteze şi pentru fiecare astfel de operator aflat pe o pozitie *k*, *T[i][j]* se incrementează cu valoarea *T[i][k-1]·T[k+1][j]*. Dacă nu se gaseşte niciun *'+'* neinclus între paranteze, atunci se caută un *'*'* şi se realizează aceleaşi acţiuni ca şi în cazul *'+'*.

Problema 7-4. Fracţii (Olimpiada Naţională de informatică, România 2001)

O proprietate interesantă a fracţiilor ireductibile este că orice fracţie se poate obţine după următoarele reguli:

- pe primul nivel se află fracţia *1/1*
- pe al doilea nivel, în stânga fracţiei *1/1* de pe primul nivel, plasăm fracţia *1/2*, iar în dreapta ei, fracţia *2/1*
- pe fiecare nivel *k* se plasează, sub fiecare fracţie *i/j* de pe nivelul de deasupra, fracţia *i/(i+j)* în stânga şi fracţia *(i+j)/j* în dreapta.

nivelul *1*: *1/1*
nivelul *2*: *1/2* *2/1*
nivelul *3*: *1/3* *3/2* *2/3* *3/1*

Dându-se o fracţie oarecare prin numărătorul şi numitorul său, determinaţi numărul nivelului pe care se află fracţia sau o fracţie echivalentă (având aceeaşi valoare) cu aceasta.

Exemple:
13/8 => nivelul 6
12/8 => nivelul 3

Soluţie: Pentru început, vom reduce fracţia *A/B* dată la o fracţie ireductibilă. Calculăm *d*, cel mai mare divizor comun al lui *A* şi *B*, apoi calculăm *P=A/d* şi *Q=B/d*. Acum trebuie să calculăm nivelul pe care se află fracţia *P/Q*. Să observăm că următorul algoritm calculează corect nivelul:

nivel=1
cât timp *P≠Q* **execută**
  dacă *P>Q* **atunci** *P=P-Q*
  altfel *Q=Q-P*
  nivel=nivel+1

Totuşi, să considerăm următorul caz: *P=2.000.000.000*, *Q=1*. În acest caz, algoritmul va executa *2.000.000.000* de iteraţii pentru a calcula corect nivelul, depăşind astfel limita de timp. Algoritmul poate fi îmbunătăţit, după cum urmează:

nivel=1
cât timp *P≠Q* **execută**
  dacă *P>Q* **atunci**
    nivel=nivel+(P div Q)
    P=P mod Q
  altfel
    nivel=nivel+(Q div P)
    Q=Q mod P

Complexitatea algoritmului este identică cu cea a algoritmului de calcul a celui mai mare divizor comun a două numere.

Problema 7-5. Codificare (SGU)

Se dă funcţia *phi(W)* a unui şir *W*:

- dacă lungimea lui *W* este *1*, atunci *phi(W)=W*
- fie $W=w_1w_2...w_N$ şi *K=N/2* (parte întreagă inferioară) => $phi(W) = phi(w_Nw_{N-1}...w_{K+1})+$ $phi(w_Kw_{K-1}...w_1)$ (unde + reprezintă concatenarea)

Exemple: phi(„Ok")="kO"; phi(„abcd")="cdab"

Dându-se lungimea N ($1 \le N \le 10^9$) a şirului W, determinaţi poziţia pe care ajunge caracterul w_q ($1 \le q \le N$) din W în *phi(W)*.

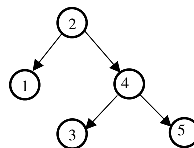
Soluţie: Vom iniţializa valoare p a poziţiei la 0, apoi vom apela următoare funcţie recursivă, cu parametrii N şi Q. La sfârşitul execuţiei, p va conţine valoarea poziţiei pe care ajunge litera w_q din W în *phi(W)*.

<u>Compute(N,Q):</u>
dacă *(N=1)* **atunci** *p=p+1*
altfel
 K=N/2
 dacă *(q≤K)* **atunci**
  p=p+N-K
  Compute*(K, K-Q+1)*
 altfel
  Compute*(N-K, N-K-(Q-K)+1)*

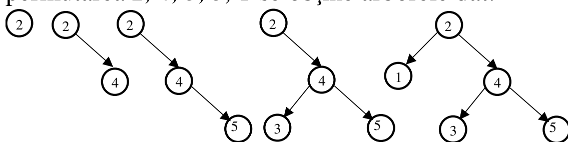
Problema 7-6. Arbore de căutare (Lotul Naţional de Informatică, România, 1999)

Se consideră un arbore binar de căutare A având n noduri conţinând cheile $1,2,\ldots,n$. O permutare $p = [p_1,\ldots,p_n]$ a numerelor întregi $1,2,\ldots,n$ se numeşte *consistentă* cu arborele A dacă arborele de căutare poate fi construit pornind de la arborele vid prin inserarea numerelor întregi p_1, p_2, …, p_n în această ordine. Să se determine numărul permutărilor mulţimii $\{1,2,\ldots,n\}$ care sunt consistente cu un arbore dat. (Altfel spus: câte secvenţe distincte de chei există pentru ca arborii binari de căutare – obţinuţi prin inserarea cheilor în ordinea dată – să fie identici cu arborele dat?)

Exemplu



Arborele din figură are exact *8* permutări consistente. De exemplu, permutările *2, 1, 4, 3, 5* şi *2, 4, 5, 3, 1* sunt consistente cu arborele din figură. Inserând elemente consecutive din permutarea *2, 4, 5, 3, 1* se obţine arborele dat:



Soluţie: Se parcurge arborele de jos în sus. Se calculează, pentru fiecare nod x, numărul de permutări posibile $NP(x)$ care produc subarborele al cărui vârf este nodul respectiv. Numărul corespunzator rădăcinii arborelui este numărul cautat. Pentru a determina câte permutări există care produc subarborele cu vârful p se procedează astfel:

1) Dacă p este nod terminal, $NP(p)=1$.
2) Dacă p nu este nod terminal şi are un singur fiu p_1, atunci $NP(p)=NP(p_1)$.
3) Dacă p nu este nod terminal şi are *2* fii, atunci fie p_1 si p_2 fiii săi. Numărul de permutări corespunzătoare subarborelui de vârf p este numărul de permutări rezultate în urma interclasării oricăror *2* permutări corespunzătoare vârfurilor p_1 şi p_2. Fiecare element al unei permutări (din subarborii lui p_1 sau p_2) îşi păstrează poziţia relativă faţă de celelalte elemente ale permutării corespunzătoare subarborelui din care face parte. Numărul de posibilităţi de a interclasa *2* şiruri de lungime a, respectiv b, este $C(a+b,a)$ (combinări de $a+b$ luate câte a). Fiecare permutare a subarborilor lui p_1 sau p_2 are $nv(p_1)$, respectiv $nv(p_2)$ elemente

($nv(x)$=numărul de noduri din subarborele cu rădăcina x; $nv(x)=1$, dacă x e nod terminal; $nv(x)=1+nv(p_1)$, dacă x are un singur fiu p_1; $nv(x)=1+nv(p_1)+nv(p_2)$, dacă x are doi fii, p_1 şi p_2). Aşadar, $NP(p)=NP(p_1)\cdot NP(p_2)\cdot C(nv(p_1)+nv(p_2), nv(p_1))$. Elementul din vârful p trebuie să se afle tot timpul înaintea oricărui element din subarborii fiilor săi în cadrul unei permutări consistente.

Complexitatea algoritmului este $O(N^2\cdot NrMari(N))$, unde $NrMari(N)$ este complexitatea de a efectua adunări cu numere mari având lungime $O(N)$ (de obicei, $NrMari(N)=O(N)$ şi, astfel, complexitatea devine $O(N^3)$).

Problema 7-7. Următorul şir de paranteze (SGU)

Se dă un şir de paranteze deschise închise corect (de lungime cel mult *10.000*). Determinaţi următorul şir de paranteze închise corect, în ordine lexicografică, care are aceeaşi lungime ca şi şirul dat. Considerăm că o paranteză deschisă *(* este mai mică lexicografic decât o paranteză închisă.

Exemplu: Pentru şirul *„(())()”*, următorul şir este *„()(())”*.

Soluţie: Fie N lungimea şirului dat. Este evident că el conţine $N/2$ paranteze deschise şi $N/2$ paranteze închise. Vom parcurge şirul de la sfârşit către început şi vom menţine două valori: nd şi ni, numărul de paranteze deschise şi închise întâlnite până la momentul respectiv în cadrul parcurgerii (iniţial, cele două valori sunt 0). Să presupunem că am ajuns la o poziţie i. Dacă pe poziţia i este o paranteză deschisă, incrementăm nd cu 1; altfel, incrementăm ni cu 1. Dacă pe poziţia i este o paranteză deschisă, vom verifica dacă există un şir care să aibă primele $i-1$ caractere ca şi şirul dat, iar pe poziţia i să aibă o paranteză închisă. Între primele $i-1$ caractere există ($N/2-nd$) paranteze deschise şi ($N/2-ni$) paranteze închise. Dacă $(N/2-nd)>(N/2-ni)$, atunci putem pune pe poziţia i o paranteză închisă. În şir mai trebuie să adăugăm, pe poziţiile $i+1, ..., N$, $nd'=nd$ paranteze deschise şi $ni'=ni-1$ paranteze închise. Vom adăuga întâi cele nd' paranteze închise, urmate de cele ni' paranteze închise. În felul acesta, am obţinut următorul şir în ordine lexicografică.

Să extindem un pic problema şi să presupunem că şirul conţine K tipuri de paranteze (de exemplu, paranteze rotunde, pătrate, acolade, etc.) şi vrem să determinăm următorul şir în ordine lexicografică care este parantezat corect. Vom presupune că există o ordine $o(1), ..., o(2\cdot K)$ ale celor $2\cdot K$ tipuri de paranteze (K tipuri de paranteze, deschise şi închise). Vom parcurge şirul de la 1 la N şi pentru fiecare poziţie i vom calcula $nd(i,j)$=câte paranteze deschise de tipul j sunt pe poziţiile $1,...,i$ şi $ni(i,j)$=câte paranteze închise de tipul j sunt pe poziţiile $1,...,i$. În general, aceste valori sunt identice cu cele de la poziţia $i-1$, cu o singură excepţie: dacă pe poziţia i e o paranteză deschisă (închisă) de tipul j, atunci avem $nd(i,j)=nd(i-1,j)+1$ ($ni(i,j)=ni(i-1,j)+1$).

Vom parcurge apoi şirul de la sfârşit către început. Pentru fiecare poziţie i, vom efectua următoarele acţiuni. Să presupunem că pe această poziţie se află caracterul cu numărul de ordine q în cadrul alfabetului de $2\cdot K$ caractere. Vom încerca să punem pe poziţia i caracterul cu cel mai mic număr de ordine $q'>q$ pentru care nu se obţin contradicţii. O contradicţie se obţine doar dacă caracterul q' este o paranteză închisă de tipul j şi $nd(i,j)<ni(i,j)+1$. Dacă nu se obţine nicio contradicţie, vom amplasa pe poziţia i caracterul q'. În continuare, vom menţine contoarele $ndc(j)$ şi $nic(j)$, reprezentând numărul curent de paranteze deschise, respectiv închise, de tipul j. Iniţializăm $ndc(j)=nd(i-1,j)$ şi $nic(j)=ni(i-1,j)$. Mai mult, dacă pe poziţia i am amplasat o paranteză deschisă (închisă) de tipul j, incrementăm $ncd(j)$ ($nic(j)$) cu

1. Acum urmează să amplasăm caracterele de pe poziţiile *i+1, ..., N*, în ordine lexicografică minimă. Avem aici două variante:

În prima variantă suntem obligaţi să păstrăm acelaşi număr de paranteze din fiecare tip ca şi în şirul iniţial. Aşadar, pentru fiecare caracter q ($1 \le q \le 2 \cdot K$), avem un număr de apariţii al său *nap(q)* care mai trebuie amplasate în şir. Pe fiecare poziţie p (de la $i+1$ la N) vom alege caracterul q minim, astfel încât $nap(q)>0$ şi nu se obţin contradicţii (se obţine o contradicţie dacă caracterul q este paranteză închisă de tipul j şi $ndc(j)<nic(j)+1$). După amplasarea sa, decremntăm cu *1* *nap(q)* şi incrementăm cu *1* *ndc(j)* (*nic(j)*) dacă caracterul q este o paranteză deschisă (închisă) de tipul j. În acest caz, atunci când selectăm prima paranteză *q'* de amplasat pe poziţia i, trebuie ca nu toate apariţiile lui *q'* să se afle între poziţiile *1,...,i-1*.

În a doua variantă, nu suntem obligaţi să păstrăm acelaşi număr de paranteze din fiecare tip ca şi în şirul iniţial. Vom reduce acest caz la cazul precedent. Pentru fiecare tip de paranteze j, dacă $ndc(j)>nic(j)$, va trebui să avem neapărat *nap(q)* caractere cu numărul de ordine q, unde acest caracter este paranteza închisă de tipul j. Pe lângă aceste caractere ale căror apariţii sunt forţate, mai avem de amplasat încă $2 \cdot h \ge 0$ caractere (obţinut ca diferenţă dintre $N-i$ şi numărul total de apariţii forţate). Vom alege caracterul cu cel mai mic număr de ordine q ce reprezintă o paranteză deschisă. Fie *q'* caracterul ce reprezintă paranteza închisă corespunzătoare. Vom incrementa *nap(q)* şi *nap(q')* cu câte h, apoi vom aplica algoritmul descris anterior.

Complexitatea soluţiei prezentate este $O(N \cdot K)$.

Problema 7-8. Umplerea unui pătrat NxN cu număr maxim de dreptunghiuri 1xK (TIMUS)

Se consideră un pătrat alcătuit din *NxN* ($1 \le N \le 10.000$) pătrăţele unitare. Determinaţi numărul maxim de dreptunghiuri de dimensiuni *1xK* (sau *Kx1*) ($1 \le K \le N$) care pot fi amplasate fără suprapuneri în interiorul pătratului (fiecare dreptunghi ocupă K pătrăţele unitare din interiorul pătratului)

Exemplu: N=5, K=4 => Răspunsul este 6.

Soluţie: Vom defini funcţia *NrMax(N,K)*, al cărei rezultat este chiar răspunsul dorit. Dacă *(N mod K)=0*, atunci *NrMax(N,K)* întoarce *((N·N) div K)* (se pot ocupa toate pătrăţelele unitare). Altfel, vom calcula două valori. $V_1=(N \bmod K) \cdot (N \text{ div } K) \cdot 4 + NrMax(N-2 \cdot (N \bmod K),K)$. Această valoare corespunde cazului în care se bordează pătratul cu un contur de „grosime” *(N mod K)*, format din dreptunghiuri de dimensiune *1xK* şi *Kx1*, iar apoi se reapelează funcţia *NrMax* pentru zona rămasă liberă (un pătrat de latură *N-2·(N mod K)*).

A doua valoare este $V_2=(N+(N \bmod K)) \cdot (N \text{ div } K)$. Acest caz corespunde amplasării a *(N div K)* dreptunghiuri pe fiecare linie a pătratului. Pe fiecare din ultimele *(N mod K)* coloane rămase libere se amplasează, din nou, câte *(N div K)* dreptunghiuri. Valoarea întoarsă de *NrMax(N,K)* este $\max\{V_1,V_2\}$.

Există şi o soluţie mai eficientă, care funcţionează pentru dreptunghiuri de dimensiuni arbitrare *PxQ* (nu doar pentru pătrate *NxN*). Vom considera cele P linii împărţite în *CP=P div K* grupuri de câte K linii (plus, eventual, încă un grup, ce conţine *RP=P mod K* linii, dacă *P mod K>0*). În fiecare grup de linii vom numerota liniile de la *0* la *K-1*. Pentru fiecare linie i, pe coloana C ($0 \le C \le Q-1$) vom considera (conceptual) că este scris numărul *(i+C) mod K*. Va trebui să numărăm în câte elemente este scris numărul *K-1*. Dacă *K=1*, acest număr este $K \cdot Q$. Altfel, pe fiecare linie există *CQ=Q div K* intervale întregi de numere consecutive *0, 1, ..., K-1*. Astfel, avem cel puţin $K \cdot CQ$ elemente egale cu *K-1*. Fie *RQ=Q mod K*. Dacă *RQ>0*,

atunci primul element egal cu $K-1$ pe una din ultimele RQ coloane ale unei linii din grup apare abia pe linia $K-1-RQ$. Pe următoarele $RQ-1$ linii va apărea elementul $K-1$ în continuare printre ultimele RQ coloane. Astfel, între primele $K-1$ linii din grup, elementul $K-1$ apare în ultimele RQ coloane de RQ ori. Aşadar, numărul total de apariţii ale elementului $K-1$ într-un grup este $K \cdot CQ+RQ$.

Vrem să calculăm acum de câte ori apare elementul $K-1$ între ultimele $RP<K$ linii. Evident, acesta apare de cel puţin $RP \cdot CQ$ ori. Dacă $RP \geq K-RQ$, atunci elementul $K-1$ mai apare de încă $RP-(K-RQ)+1$ ori pe ultimele RQ coloane ale ultimelor RP linii. Numărul total de apariţii ale elementului $K-1$ este egal chiar cu numărul maxim de dreptunghiuri $1xK$ (sau $Kx1$) ce pot fi amplasate în interiorul dreptunghiului dat. Observăm că algoritmul descris are ordinul de complexitate $O(1)$ (efectuând doar $O(1)$ împărţiri, adunări şi înmulţiri), sau $O(log(P)+log(Q))$ (dacă numerele P, Q şi/sau K sunt numere mari).

Problema 7-9. Şiruri cu intervale cu lungimi minime şi maxime

Determinaţi numărul de şiruri de lungime N, în care fiecare element ia valori din mulţimea $\{0,1,...,B-1\}$ şi în care nu există mai mult de K ($0 \leq K \leq N$) elemente de 0 consecutive, dar există cel puţin X şi cel mult Y ($0 \leq X \leq Y \leq N$) secvenţe de cel puţin Q zerouri consecutive ($0 \leq Q \leq K$). O secvenţă este un interval maximal ca număr de 0-uri. Găsiţi un algoritm eficient pentru $Q=0$ (când valorile lui X şi Y nu contează).

Soluţie: Vom calcula $NSC(i,j,p)$=numărul de şiruri de lungime i care se termină cu j 0-uri şi care conţin p secvenţe de cel puţin Q şi cel mult K 0-uri consecutive (exclusiv ultima secvenţă de j 0-uri) ($0 \leq i \leq N$, $0 \leq j \leq i$, $0 \leq p \leq Y$). Avem $NSC(0,0,0)=1$. Pentru $i \geq 1$ avem: $NSC(i,0,p)=(B-1) \cdot (NSC(i-1,0,p)+NSC(i-1,1,p)+...+NSC(i-1,min\{i-1,Q-1\},p))$ şi $NSC(i,j \geq 1,p)=NSC(i-1,j-1,p)$.

Pentru $p \geq 1$ şi $j=0$, vom adăuga la valoarea $NSC(i,j,p)$ calculată conform formulelor anterioare, valoarea $(B-1) \cdot (NSC(i-1,Q,p-1)+NSC(i-1,Q+1,p-1)+...+NSC(i-1,min\{i,K\},p-1))$. Rezultatul este suma valorilor $NSC(N, 0 \leq j \leq Q-1, X \leq p \leq Y)$, la care adăugăm suma valorilor $NSC(N, Q \leq j \leq K, X-1 \leq p \leq Y-1)$. Complexitatea unui algoritm care implementează aceste formule este $O(N^3)$.

Pentru cazul $Q=0$, vom calcula $NS(i)$=numărul de şiruri de lungime i, ce conţin cel mult K elemente de 0 consecutive şi care se termină cu un element diferit de 0. $NS(0)=1$. Vom considera fiecare lungime i ($1 \leq i \leq N$), în ordine crescătoare. $NS(i)=(B-1) \cdot (NS(i-1)+NS(i-2)+...+NS(max\{0,i-(K+1)\}))$. O implementare directă ar avea complexitatea $O(N^2) \cdot NumereMari(N)$. Pentru a ajunge la complexitatea $O(N) \cdot NumereMari(N)$, vom menţine o sumă SNS a ultimelor cel mult $K+1$ valori $NS(j)$ calculate. Iniţializăm SNS cu $NS(0)$. Apoi, pentru fiecare lungime i ($1 \leq i \leq N$), avem $NS(i)=(B-1) \cdot SNS$. După aceea, adăugăm $NS(i)$ la SNS, iar dacă $i-(K+1) \geq 0$, setăm $SNS=SNS-NS(i-(K+1))$. Rezultatul final este $NS(N-0)+NS(N-1)+...+NS(N-K)$ (variem lungimea $j=0,...,K$ a ultimei secvenţe de elemente 0).

Dacă lungimea şirurilor ar fi foarte mare (de ex., 10^9), K-ul ar fi mai mic (de ex., $0 \leq K \leq 100$) şi am avea nevoie doar de rezultat *modulo* un număr P, atunci am putea folosi următoarea metodă. Am defini un vector-coloană $x(i)$, ce constă din $K+1$ elemente. Elementele acestui vector sunt, în ordine, $NS(i), NS(i-1), ..., NS(i-K)$. Vom defini o matrice de transformare T, având $K+1$ linii şi coloane. Prima linie a lui T conţine numai valori de $(B-1)$, iar pe următoarele linii i ($2 \leq i \leq K+1$) avem $T(i,i)=1$ şi $T(i,j \neq i)=0$. Observăm că, pentru a calcula $x(i)$, putem înmulţi matricea T cu vectorul-coloană $x(i-1)$. Dacă pornim de la vectorul $x(0)=(1, 0, 0, ..., 0)$, putem scrie $x(i)=T^i \cdot x(0)$. Rezultatul dorit este suma elementelor

vectorului $x(N)$. Pentru a calcula $x(N)$ este suficient să ridicăm la puterea N matricea T şi să înmulţim T^N cu $x(0)$. Putem realiza acest lucru în timp logaritmic – folosind $O(log(N))$ înmulţiri de matrici (deci putem obţine o complexitate de $O((K+1)^3 \cdot log(N))$. Toate operaţiile (de adunare şi înmulţire) se realizează modulo P.

Probleme asemănătoare au fost propuse la diverse concursuri, precum concursuri online de pe TIMUS sau baraje din cadrul lotului naţional de informatică.

Problema 7-10. Ture (ACM ICPC NEERC 2004, Southern Subregion)

Se dă o tablă de şah mai specială. Aceasta are N ($1 \le N \le 300$) linii, dar fiecare linie i are un număr de coloane $C(i)$ ($1 \le C(i) \le 300$) (coloanele de pe această linie sunt coloanele $1, 2, ..., C(i)$). Determinaţi în câte moduri pot fi amplasate K ture pe tabla de şah, în aşa fel încât oricare două ture să nu se atace. Două ture se atacă dacă sunt amplasate pe aceeaşi linie sau aceeaşi coloană. Două ture de pe aceeaşi coloană j se atacă chiar dacă între liniile lor există linii i cu $C(i)<j$.

Exemplu: N=2, K=2, C(1)=2, C(2)=3 => Răspunsul este 4.

Soluţie: Pe o tablă de şah dreptunghiulară cu P linii şi Q coloane se pot amplasa K ture în $C(P,K) \cdot C(Q,K) \cdot K!$ moduri ($C(x,y)$=combinări de x luate câte y), deoarece liniile pe care se aleg aceste ture se pot alege în $C(P,K)$ moduri şi, independent, coloanele se pot alege în $C(Q,K)$ moduri; pentru K linii şi coloane alese există $K!$ moduri de a amplasa cele K ture.

Pentru o tablă în care nu au toate liniile acelaşi număr de coloane, vom sorta liniile crescător după numărul de coloane, astfel încât să avem $C(1) \le C(2) \le .. \le C(N)$. Apoi vom calcula valorile $NT(i,j)$=în câte moduri se pot amplasa, fără să se atace, j ture pe primele i linii. Avem $NT(0,0)=1$ şi $NT(0,j>0)=0$. Pentru $1 \le i \le N$ avem: $NT(i, 0)=1$, $NT(i, 1 \le j \le min\{i,K,C(i)\})=NT(i-1, j)+NT(i-1,j-1) \cdot (C(i)-(j-1))$. Primul termen corespunde cazului când nu amplasăm nicio tură pe linia i, iar al doilea termen corespunde cazului când amplasăm o tură pe linia i.

Dacă avem j ture în total, atunci $(j-1)$ ture au fost amplasate pe liniile $1,...,i-1$ şi acestea „ocupă” $j-1$ coloane dintre cele $C(i)$ ale liniei i. Prin urmare, tura de pe linia i poate fi amplasată pe una din cele $(C(i)-(j-1))$ coloane „libere”.

Problema 7-11. Coliere frumoase (SGU)

Să considerăm un colier alcătuit din $2 \cdot N-1$ ($5 \le 2 \cdot N-1 \le 2^{31}-1$) perle, dintre care K perle sunt negre (celelalte fiind albe). Un colier este *frumos* dacă putem alege două perle negre (nu neapărat diferite) în aşa fel încât una dintre cele două părţi de colier dintre cele două perle să conţină exact N perle (indiferent ce culori au acestea). Determinaţi numărul K minim pentru care fiecare colier format din $2 \cdot N-1$ perle este *frumos*.

Exemple:

2·N-1=5 => Valoarea minimă pentru K este 3.

2·N-1=7 => Valoarea minimă pentru K este 4.

Soluţie: Din numărul $X=2 \cdot N-1$ dat calculăm valoarea lui N ($N=(X+1)/2$). Dacă $(N \bmod 3)=2$, atunci valoarea minimă a lui K este $N-1$; altfel, valoarea minimă a lui K este N.

Problema 7-12. Cicluri hamiltoniene în grafuri multipartite (TIMUS)

Se dă un graf multipartit complet. Graful conţine N ($2 \le N \le 5$) părţi . Fiecare parte i ($1 \le i \le N$) conţine $B(i)$ ($1 \le B(i) \le 30$) noduri. Graful este format în felul următor: există câte o

muchie între oricare două noduri situate în părţi diferite şi nu există nicio muchie între nodurile din aceeaşi parte. Determinaţi numărul de cicluri Hamiltoniene din graf (*modulo* un număr prim *P*; $3 \le P \le 100.000$). Nodurile se consideră numerotate şi avem $(1+max\{B(i)\})^N \le 5.000.000$.

Exemplu: N=3, B(1)=B(2)=B(3)=2, P=997 => Răspunsul este 16.

Soluţie: Vom considera că ciclul începe într-unul din nodurile din prima parte. Pentru început vom considera că ordinea în care sunt traversate pe ciclu nodurile din aceeaşi parte este fixată. Astfel, ciclul poate fi descris prin numărul de noduri din fiecare parte prin care s-a trecut şi de partea curentă la care s-a ajuns. Vom calcula $NC(x(1), ..., x(N), k)$=numărul de drumuri care încep la primul nod din partea *1*, trec prin $x(i)$ noduri din partea *i* ($0 \le x(i) \le B(i)$) şi ajung în partea *k* ($1 \le k \le N$).

Iniţial, avem $NC(1, 0, ..., 0, 1)=1$ şi celelalte valori sunt iniţializate la *0*. Vom considera tuplurile $(x(1), ..., x(N))$ în ordine crescătoare a sumei ($x(1)+...+x(N)$) (de la suma *2* încolo). Tuplurile cu aceeaşi sumă pot fi considerate în orice ordine. Pentru un tuplu $(x(1), ..., x(N))$ şi o valoare *k* ($1 \le k \le N$ şi $x(k) \ge 1$), $NC(x(1), ..., x(N), k)$ este egal cu suma valorilor $NC(x(1), ..., x(k-1), x(k)-1, x(k+1), ..., x(N), j)$ ($1 \le j \le N$, $j \ne k$). Numărul total de cicluri hamiltonien (cu ordinea fixată a nodurilor în cadrul fiecărei părţi) este *NH*=suma valorilor $NC(B(1), ..., B(N), j)$ ($2 \le j \le N$).

Pentru a obţine rezultatul final vom calcula $NHF=(NH \cdot (B(1)-1)! \cdot B(2)! \cdot ... \cdot B(N)!)/2$. Factorul $B(i)!$ determină toate modalităţile de ordonare a nodurilor din partea *i* în cadrul ciclului. Factorul $(B(1)-1)!$ Indică faptul că primul nod de pe ciclu rămâne fixat, dar, în cazul părţii *1*, ordinea pe ciclu a celorlalte ($B(1)-1$) noduri poate fi aleasă oricum. Împărţirea la *2* este necesară, deoarece fiecare ciclu este calculat de două ori, câte o dată pentru fiecare sens de parcurgere al său. Toate operaţiile de adunare şi înmulţire se efectuează modulo *P*. Pentru împărţirea la *2*, trebie să calculăm inversul multiplicativ al lui *2*, adică acel număr *x* cu proprietatea că $x \cdot 2=1 \ (mod\ P)$ (putem încerca fiecare valoare a lui *x*, între *1* şi *P-1*, sau putem calcula pe *x* direct, folosind algoritmul lui Euclid extins).

Problema 7-13. Combinări cu restricţii de sumă (SGU)

Se dă un număr prim *P* ($3 \le P \le 1000$). Determinaţi în câte moduri se pot alege *P* elemente distincte din mulţimea $\{1,2,...,2 \cdot P\}$, astfel încât suma lor să fie multiplu de *P*.

Exemple: N=3 => Răspunsul este 8; N=5 => Răspunsul este 52.

Soluţie: Voi prezenta întâi o soluţie generală, care nu ţine cont de faptul că *P* este număr prim. Vom calcula valorile $NP(i,j,k)$=numărul de posibilităţi de a alege *j* elemente din mulţimea $\{1,...,i\}$ ($j \le min\{i,P\}$), astfel încât suma elementor alese, *modulo P*, să fie egală cu *k* ($0 \le k \le P-1$). Avem $NP(0,0,0)=1$ şi $NP(0,j>0,*)=NP(0,*,k>0)=0$. Pentru $i \ge 1$ avem: $NP(i,j,k)=NP(i-1,j,k)+NP(i-1,j-1,(k-i+P) \ mod \ P)$. Primum termen al sumei corespunde cazului când elementul *i* nu este ales, iar al doilea termen corespunde cazului când alegem elementul *i*. Răspunsul este $NP(2 \cdot P, P, 0)$. Complexitatea acestui algoritm este $O(P^3 \cdot NumereMari(P))$. Memoria folosită pare a fi de oridnul $O(P^3)$, dar, întrucât pentru calculul valorilor $NP(i,*,*)$ avem nevoie doar de valorile $NP(i-1,*,*)$, ea poate fi redusă la $O(P^2)$.

Dacă studiem valorile $NP(2 \cdot P, P, *)$ pentru *P* număr prim, vom observa că valorile $NP(2 \cdot P, P, 1 \le j \le P-1)$ sunt egale între ele, iar $NP(2 \cdot P, P, 0)=NP(NP(2 \cdot P, P, 1)+2$. Întrucât suma tuturor acestor valori este $C(2 \cdot P, P)$ (combinări de $2 \cdot P$ luate câte *P*), obţinem că

răspunsul este $(C(2\cdot P, P)-2)/P+2$. Vom calcula $C(2\cdot P, P)$ folosind $O(P)$ operaţii pe numere mari (înmulţirea număr mare cu număr mic sau împărţirea număr mare la număr mic). Vom folosi următoarele relaţii: $C(X,0)=1$ şi $C(X,1\leq j\leq X)=(X-j+1)/j\cdot C(X,j-1)$. Cealaltă relaţie binecunoscută ($C(X,1\leq j\leq X)=C(X-1,j-1)+C(X-1,j)$) nu este folositoare în acest caz, deoarece ar conduce la un algoritm ce foloseşte $O(P^2)$ operaţii (adunări) pe numere mari.

Problema 7-14. Parantezări cu adâncime dată (Olimpiada Baltică de Informatică 1999)

Determinaţi câte şiruri ce conţin N ($1\leq N\leq 38$) perechi de paranteze închise corect există, astfel încât adâncimea şirului să fie D ($1\leq D\leq N$). Adâncimea $D(S)$ a unui şir S de paranteze se defineşte în felul următor:

- dacă S este şirul vid, $D(S)=0$
- dacă S e de forma (S'), unde S' este un şir de paranteze, $D(S)=1+D(S')$
- dacă $S=AB$, unde A şi B sunt două şiruri de paranteze, atunci $D(S)=max\{D(A), D(B)\}$

Exemplu: N=3, D=2 => Răspunsul este 3. Şirurile de adâncime 2 sunt: (())() ; ()(()) ; (()())

Soluţie: Vom calcula $NS(i,j)$=numărul de şiruri formate din i perechi de paranteze închise corect, având adâncime j. Avem: $NS(i\geq 0,j>i)=0$, $NS(0,0)=1$ şi $NS(i\geq 1, 1)=1$.

Pentru $i\geq 2$ şi $2\leq j\leq i$ avem următoarele posibilităţi. Şirul poate fi format dintr-un şir A ce conţine k perechi de paranteze, având adâncime $j-1$, pe care îl includem între o altă pereche de paranteze; acest şir este urmat de un şir B format din $i-k-1$ perechi de paranteze şi orice adâncime $l\leq j$. Vom nota prin $N_1(i,j)$=suma valorilor $(NS(k, j-1)\cdot NS(i-k-1,l))$ ($0\leq k\leq i-1$; $0\leq l\leq j$).

O a doua posibilitate este ca şirul de i perechi de paranteze şi adâncime j să fie format dintr-un şir A format din k perechi de paranteze şi adâncime $l<j-1$, pe care îl includem într-o altă pereche de paranteze, urmat de un şir B format din $(i-k-1)$ paranteze şi adâncime j. Vom nota cu $N_2(i,j)$ suma valorilor $(NS(k,l)\cdot NS(i-k-1,j))$ ($0\leq k\leq i-1$; $0\leq l\leq j-2$). Avem apoi $NS(i,j)=N_1(i,j)+N_2(i,j)$.

Dacă aplicăm formulele direct, obţinem un algoritm cu complexitatea $O(N^4)$. Putem reduce complexitatea la $O(N^3)$ în felul următor. După ce am calculat toate valorile $NS(q,*)$, vom calcula nişte sume prefix: $SNS(q,-1)=0$ şi $SNS(q,j\geq 0)=SNS(q,j-1)+NS(q,j)$. Apoi, în cadrul formulelor de calcul pentru $N_1(i,j)$ şi $N_2(i,j)$, vom considera, pe rând, fircare valoare a lui k. După fixarea lui k, pentru N_1 trebuie să înmulţim valoarea $NS(k,j-1)$ (fixată) cu suma valorilor $NS(i-k-1,l)$ ($0\leq l\leq j$); această sumă este egală cu $SNS(i-k-1,j)$. Pentru calculul lui $N_2(i,j)$, după alegerea valorii lui k, trebuie să înmulţim $NS(i-k-1,j)$ cu suma valorilor $NS(k,l)$ ($0\leq l\leq j-2$); această sumă este egală cu $SNS(k,j-2)$. Astfel, fiecare valoare $N_1(i,j)$, $N_2(i,j)$ şi $NS(i,j)$ se calculează în timp $O(N)$, complexitatea totală devenind $O(N^3)$.

Problema 7-15 Suma cifrelor (TIMUS)

Fie $S(N)$ suma cifrelor unui număr N. Determinaţi pentru câte perechi (A,B) de numere în baza Q ($2\leq Q\leq 50$) de exact K ($1\leq K\leq 50$) cifre are loc egalitatea: $S(A+B)=S(A)+S(B)$ (în baza Q). La numărare trebuie să ţineţi cont de următoarele fapte: *(1)* numerele A şi B nu pot fi 0 şi nici nu pot începe cu cifra 0; *(2)* ordinea numerelor în cadrul unei perechi contează => de ex., perechile *(12,26)* şi *(26,12)* (în baza $Q\geq 7$) sunt diferite.

Exemplu: K=2, Q=10 => Răspunsul este 1980.

Soluţie: Răspunsul este $(Q-1)\cdot(Q-2)/2\cdot((Q+1)\cdot Q/2)^{(K-1)}$. Ajungem la acest răspuns observând că atunci când adunăm cele două numere nu trebuie să avem niciodată transport. Există *(Q-*

$1)\cdot(Q-2)/2$ de perechi de cifre din mulţimea $\{1,...,Q-1\}$ care adunate nu depăşesc valoarea $Q-1$. Pentru poziţiile $2,...,K$, se poate folosi şi cifra 0, astfel că există $(Q+1)\cdot Q/2$ de perechi de cifre din mulţimea $\{0,...,Q-1\}$ pentru care nu se obţine transport la adunare.

Problema 7-16. Anticip (Lotul Naţional de Informatică, România 2005)

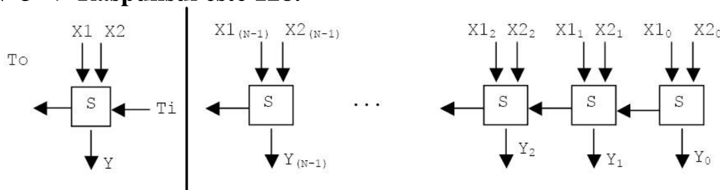
Un sumator pe un bit este un mic dispozitiv cu *3* intrări şi *2* ieşiri. El primeşte la intrare X_1, X_2 şi T_i. X_1 şi X_2 sunt biţii ce trebuie adunaţi, iar T_i este transportul anterior (ca intrare). La ieşire furnizează Y şi T_o. Y este suma, iar T_o este transportul următor (ca ieşire). Pentru a formaliza aceste lucruri putem scrie:

- $Y = (X_1 + X_2 + T_i)\ mod\ 2$ (*mod* este restul împărţirii întregi)
- $T_o = (X_1 + X_2 + T_i)\ div\ 2$ (*div* este câtul împărţirii întregi)

Pentru a aduna numere pe *N* biţi se folosesc *N* astfel de sumatoare. Ele sunt legate ca în figura de mai jos, adică transportul de ieşire al unui sumator este transportul de intrare pentru următorul. Problema cu aceste sumatoare pe mai multi biţi este că un sumator trebuie să aştepte transportul de la unitatea anterioară (exceptând primul sumator). Dacă un sumator pe un bit face calculul într-o secundă, atunci pentru un sumator pe *N* biţi (format din *N* sumatoare pe un bit) vor fi necesare *N* secunde. Pentru a îmbunătăţi performanţa acestor sumatoare pe *N* biţi s-au introdus nişte unităţi capabile să anticipeze transportul, adică intrarea T_i. Aceste unităţi verifică datele de intrare precedente $X_1(i-1)$ şi $X_2(i-1)$. Dacă amândouă sunt *0* atunci T_i va fi *0*, indiferent de ce primeşte acea unitate ca transport de intrare. De asemenea, dacă amândouă sunt *1*, atunci T_i va fi *1*. Toate sumatoarele care folosind anticipaţia pot calcula transportul de la sumatorul precedent încep calculul odată cu primul sumator. Comunicarea transportului de la un sumator la următorul se realizează instantaneu.

Cercetătorii care au inventat aceste unităţi de transport vor să ştie cu cât îmbunătăţesc performanţa sistemului şi au hotărât să se facă toate adunările posibile, pentru a compara cu vechiul sistem. Prin toate adunările posibile se înţelege că se va aduna orice număr pe *N* biţi cu orice număr pe *N* biţi fix o dată (în total 4^N adunări). Ei vor să ştie cât au durat aceste operaţii, adică suma tuturor timpilor pentru fiecare adunare.

Exemplu: N=3 => Răspunsul este 128.



Soluţie: Observăm că o adunare se împarte în mai multe blocuri continue (în funcţie de unii sumatori care anticipează transportul) care se efectuează în paralel (adică fiecare bloc începe adunarea în acelaşi timp). Timpul de execuţie al unei adunări este deci maximul dintre lungimile blocurilor care se formează. Construim următoarea matrice:

- $A[i][j][k]$ = câte posibilităţi de a aduna primii *i* biţi, cu timpul maxim *j* (adică lungimea maximă a unui bloc) şi ultimul bloc de lungime *k* (bloc care poate continua), pentru $k \geq 1$
- $A[i][j][0]$ = câte posibilităţi de a aduna primii *i* biţi, cu timpul maxim *j* (adică lungimea maximă a unui bloc) şi ultimul bloc se termină după sumatorul *i*)

Atunci când la bitul $i+1$ se vor aduna biţii $0+1$ sau $1+0$, deoarece aceştia nu pot fi anticipaţi, nu vor crea un nou bloc şi se va actualiza valoarea $A[i+1][max(j,k+1)][k+1]$.

Atunci când la bitul *i+1* se vor aduna biţii *0+0* sau *1+1*, deoarece aceştia vor fi anticipaţi, se va crea un nou bloc după ei şi se va actualiza valoarea *A[i+1][max(j,k+1)][0]*.

Iniţializăm *A[1][1][1] = A[1][1][0] = 2* (*2* dintre posibilităţi formează un bloc care se termină, pentru că sumatorul nu poate fi anticipat, iar celelalte *2* formează un bloc care poate continua). Deci recurenţa este următoarea:

- *A[i+1][max(j,k+1)][k+1] += 2·A[i][j][k]*
- *A[i+1][max(j,k+1)][0] += 2·A[i][j][k]*

Răspunsul cerut se va afla astfel:

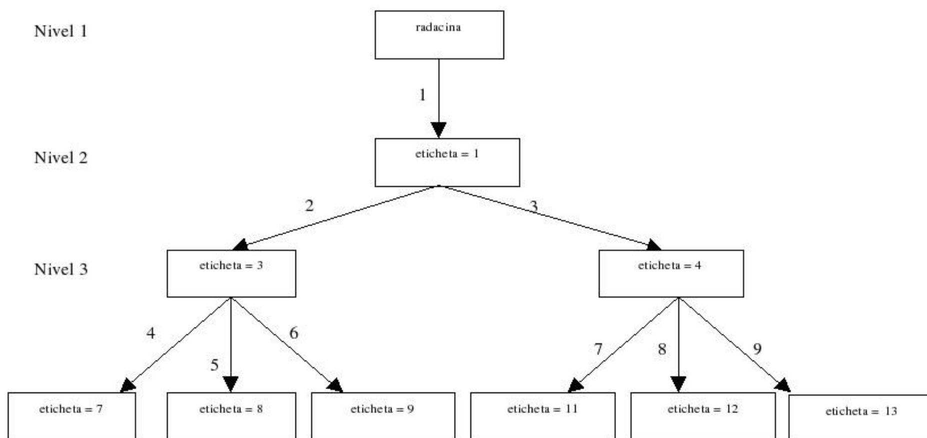
- *R += j· (A[N][j][0] + A[N][j][1] + … + A[N][j][N]), 1≤j≤N.*

Problema 7-17. J-Arbore (Happy Coding, infoarena)

J-arborele este un arbore infinit cu următoarele proprietăţi:

- pe nivelul *1* al arborelui există un singur nod (rădăcina)
- fiecare nod de pe nivelul *i* are exact *i* fii
- muchiile arborelui se etichetează cu numere întregi consecutive, mergând pe nivele de sus în jos (începând cu primul nivel al arborelui) şi pe fiecare nivel mergând de la stanga la dreapta
- toate nodurile în afara rădăcinii vor fi etichetate cu numere întregi egale cu suma muchiilor de pe drumul de la rădăcina la nodul respectiv

Mai jos aveti primele nivele ale unui astfel de arbore.



Dându-se un numar natural *X* ($1 \le X \le 10^{18}$), vi se cere să spuneţi dacă există un nod etichetat cu valoarea *X* şi să afişaţi etichetele muchiilor de pe drumul de la rădăcina spre nodul respectiv.

Exemple:
X=100 => Nu există.
X=1 => 1
X=2 => Nu există
X=12 => 1 3 8
X=17 => 1 2 4 10
X=89 => 1 2 5 16 65
X=666 => 1 3 7 22 97 536

Soluţie: Vom calcula *N[i]*, reprezentând câte noduri are arborele pe fiecare nivel *i* (începând de la *1*), precum şi valoarea de pe prima muchie şi de pe ultima muchie dintre nivelul *i-1* şi nivelul *i* ($v_1[i]$ şi $v_2[i]$). Avem *N[1]=1* şi $v_1[1]=v_2[1]=0$. În cazul general, *N[i]=N[i-1]·(i-1)*, $v_1[i]=v_2[i-1]+1$ şi $v_2[i]=v_1[i]+N[i]-1$. De asemenea, vom calcula cea mai mică şi cea mai mare etichetă a unui nod de pe nivelul *i*: $vmin[i]=vmin[i-1]+v_1[i]$ şi $vmax[i]=vmax[i-1]+v_2[i]$ (*vmin[1]=vmax[1]=0*).

Vom observa că *24* de nivele ale arborelui ajung pentru limitele date. Pentru fiecare număr *X* dat vom găsi nivelul pe care s-ar afla (dacă numărul există în arbore), folosind valorile *vmin[i]* şi *vmax[i]* (căutăm acel nivel *i* pentru care *vmin[i]≤X≤vmax[i]*; o căutare secvenţială este suficientă, deoarece există doar *24* de nivele care ne interesează pentru limitele date; totuşi, se poate utiliza şi căutarea binară, deoarece avem *vmin[i]>vmax[i-1]*). Dacă nu găsim niciun nivel cu proprietatea dorită, numărul *X* nu există ca etichetă în arbore.

Apoi, pentru a determina unde se află numărul *X*, vom încerca să determinăm poziţia acestuia in cadrul nivelului *i*. Pentru aceasta vom realiza o căutare binară. Vom scrie o funcţie care va determina ce valoare se află pe poziţia *p* de pe nivelul *q*: această valoare este *1*, dacă nivelul este *1* (deoarece există un singur nod pe nivelul *1*), sau valoarea de pe poziţia *p/q* de pe nivelul *q-1*, la care se adaugă $v_1[q]+p$ (considerând poziţiile numerotate de la *0* în cadrul unui nivel). Vom determina astfel dacă numărul *X* există şi, dacă da, vom reconstitui drumul de la el până la radacina arborelui (ceea ce facem, oricum, în cadrul funcţiei de determinare a valorii de pe o poziţie *p* din cadrul unui nivel *q*; ultima muchie are valoarea $v_1[q]+p$, după care ne mutăm pe poziţia corespunzătoare de pe nivelul de deasupra, ş.a.m.d. până ajungem pe nivelul *1*).

Problema 7-18. Numărul punctelor de intersecţie ale diagonalelor (infoarena)

Se consideră un poligon convex cu *N* (*3≤N≤100.000*) vârfuri. Ştiind că oricare trei diagonale ale poligonului nu se intersectează în acelaşi punct, determinaţi numărul total de puncte de intersecţie ale diagonalelor poligonului.

Exemplu: N=4 => Răspunsul este 1.

Soluţie: Oricare *4* vârfuri ale poligonului determină două diagonale ale poligonului care se intersectează. Deci răspunsul este *C(N,4)* (combinări de *N* luate câte *4*). O soluţie cu complexitatea *O(N)* este următoarea. Vom calcula *NP(i)*=numărul punctelor de intersecţie ale diagonalelor unui poligon cu *i* vârfuri. Vom începe cu *NP(3)=0* şi vom calcula *NP(i)*, cu *i* variind crescător de la *4* la *N*. Pentru a calcula *NP(i)* pe baza lui *NP(i-1)* va trebui să calculăm câte puncte de intersecţie noi aduce adăugarea unui nou vârf. Vârful *i* are *i-3* diagonale adiacente. Numărul de puncte de intersecţie suplimentare *NPS(i)* este suma valorilor *j·(i-2-j)* (*1≤j≤i-3*). Dacă am calcula această sumă în timp liniar, am obţine o complexitate $O(N^2)$. Putem, însă, să calculăm *NPS(i)* în *O(1)*, pe baza lui *NPS(i-1)* (începând cu *NPS(3)=0*). Avem *NPS(i)=NPS(i-1)+(i-4)·(i-3)/2+i-3*.

Problema 7-19. Numărul de zone interioare determinate de diagonalele unui poligon convex (UVA)

Se consideră un poligon convex cu *N* (*3≤N≤100.000*) vârfuri. Ştiind că oricare trei diagonale ale poligonului nu se intersectează în acelaşi punct, determinaţi numărul total de zone delimitate de diagonale în interiorul poligonului.

Exemplu: *N=4 => Răspunsul este 4.*

Soluţie: Răspunsul este *(N-1)·(N-2)·*$(N^2$*-3·N+12)/24* (demonstraţi !).

Problema 7-20. Codificare numerică a unui şir de paranteze (UVA)

Întrucât şirurile formate din paranteze sunt greu de urmărit cu ochiul liber, s-a propus înlocuirea parantezelor prin numere. Astfel, o pereche de paranteze care se închid reciproc sunt înlocuite în şir prin acelaşi număr x. De exemplu, şirul *(()(()))* poate fi scris ca *1 2 2 1 3 3 1 1*. Vedem, însă, imediat, că şirul *1 2 2 1 3 3 1 1* nu are o interpretare unică că şir de paranteze. De exemplu, el ar putea proveni şi de la şirul de paranteze *(())()()*. Pentru un şir de numere dat, determinaţi dacă există mai mult de *K≥1* interpretări ale sale ca şiruri de paranteze.

Soluţie: Vom calcula *NP(i,j)*=numărul de şiruri de paranteze din care poate rezulta subsecvenţa de numere dintre poziţiile *i* şi *j* din şirul *S* dat. Avem *NP(i,i-1)=1*. Vom considera perechile *(i,j)* în ordine crescătoare a lui *j-i*. Dacă *S(i)=S(j)* (numărul de pe poziţia *i* din *S* este egal cu numărul de pe poziţia *j* din *S*), atunci iniţializăm *NP(i,j)=NP(i+1, j-1)*; altfel, iniţializăm *NP(i,j)=0*. Apoi considerăm toate poziţiile *p* (*i+1≤p≤j-1*). Dacă *S(i)=S(p)*, atunci incrementăm *NP(i,j)* cu *NP(i+1,p-1)·NP(p+1, j)* (considerăm că paranteza care se deschide pe poziţia *i* se închide pe poziţia *p*). După fiecare incrementare, dacă *NP(i,j)>(K+1)*, setăm *NP(i,j)=K+1*. Dacă *NP(1,N)=K+1*, atunci există cel puţin *K+1* şiruri de paranteze care pot fi codificate folosind şirul *S* dat (*N*=lungimea şirului *S*).

Problema 7-21. Grădină liniară (Olimpiada Internaţională de Informatică, 2008, enunţ modificat)

Se consideră şiruri formate din caracterele *„L”* şi *„P”* (unde *„L”* este lexicografic mai mic decât *„P”*), cu proprietatea că diferenţa (în valoare absolută) dintre numărul de *„L”*-uri şi numărul de *„P”*-uri din orice subsecvenţă de caractere consecutive este cel mult *D* (*1≤D≤5*). Dându-se un astfel de şir cu *N* caractere (*1≤N≤1.000.000*), determinaţi al câtelea şir este (*modulo* un număr *M*) în ordine lexicografică, dintre toate şirurile care au proprietăţile menţionate.

Exemplu: şirul „PLPPL”, D=2, M=7 => Răspunsul este 5. Şirul este, de fapt, al 12-lea şir în ordine lexicografică, iar 12 mod 7=5.

Soluţie: Vom considera că unui caracter *„L”* îi corespunde valoarea *-1*, iar unui caracter *„P”* îi corespunde valoarea *+1*. Astfel, vom considera doar şiruri alcătuite din numerele *-1* şi *+1*. Vom calcula *NS(k, pmin, pmax)*=numărul de şiruri (cu elemente egale cu *+1* sau *-1*), pentru care cea mai mică sumă prefix (de la stânga la dreapta) este mai mare sau egală cu *pmin*, iar cea mai mare sumă prefix este mai mică sau egală cu *pmax*. Observăm că prima sumă prefix este întotdeauna *0*, obţinând că *pmin≤0* şi *pmax≥0*. De asemenea, din condiţiile impuse de problemă, vom avea *pmin≥-D* şi *pmax≤D*.

Pentru *k=0*, avem *NS(k, pmin, pmax)=1 mod M* (*-D≤pmin≤0*; *0≤pmax≤D*). Pentru *k≥1* şi fiecare pereche *(pmin, pmax)*, vom proceda după cum urmează. Vom iniţializa *NS(k, pmin, pmax)=0*. Apoi vom considera, pe rând, fiecare din cele *2* caractere (*-1* sau *+1*) care ar putea fi primul caracter din şir. Să presupunem că acest caracter este *Q*. Vom calcula valorile *nextpmin=pmin-Q* şi *nextpmax=pmax-Q*. Dacă *nextpmin>0* sau *nextpmax<0*, nu vom considera mai departe cazul caracterului *Q* (pentru tuplul *(k, pmin, pmax)*). Altfel, vom seta *nextpmin=max{nextpmin, -D}* şi *nextpmax=min{nextpmax, D}*. *nextpmin* şi *nextpmax*

reprezintă suma prefix minimă şi, respectiv maximă, dacă am începe calcularea sumelor prefix de la al doilea caracter încolo. Astfel, vom seta $NS(k, pmin, pmax)=(NS(k, pmin, pmax)+NS(k-1, nextpmin, nextpmax)) \bmod M$. $NS(N, -D, D)$ reprezintă numărul de şiruri de lungime N care au diferenţa (în valoare absolută) dintre numărul de *„L”*-uri şi numărul de *„P”*-uri din fiecare subsecvenţă continuă, cel mult egală cu D.

O generalizare a problemei s-ar fi putut realiza dacă impuneam ca diferenţa dintre numărul de *„L”*-uri şi numărul de *„P”*-uri din orice subsecvenţă continuă să fie cel mult LP, iar cea dintre numărul de *„P”*-uri şi cel de *„L”*-uri să fie cel mult PL. În algoritmul folosit am fi calculat $pmin$ între $-LP$ şi 0, $pmax$ între 0 şi PL, iar înaintea de mărirea valorii lui $NS(k, pmin, pmax)$, am fi înlocuit instrucţiunile $nextpmin=max\{nextpmin, -D\}$ cu $nextpmin= max\{nextpmin, -LP\}$, şi $nextpmax=min\{nextpmax, D\}$, cu $nextpmax=max\{nextpmax, PL\}$. $NS(N, -LP, PL)$ ar fi fost numărul de şiruri de lungime N cu proprietăţile impuse.

Să vedem acum cum vom determina, pentru un şir $S(1), ..., S(N)$ dat, al câtelea şir în ordine lexicografică este. Vom menţine un contor cnt (iniţializat la 0), care, la sfârşit (după parcurgerea şirului S), va reprezenta numărul de şiruri mai mici din punct de vedere lexicografic decât şirul S. Aşadar, răspunsul va fi $(cnt+1) \bmod M$. Vom menţine şi valorile p, $pmin$ şi $pmax$. p reprezintă suma prefix curentă, iar $pmin$ şi $pmax$ reprezintă suma minimă, respectiv maximă, a unei sume sufix, calculată de la caracterul curent al şirului, spre stânga. Iniţial, $p=pmin=pmax=0$. În continuare, vom parcurge şirul de la stânga la dreapta, cu o variabilă i ($1 \le i \le N$).

Să presupunem că am ajuns la poziţia i. Dacă $S(i)=$*”L”*, atunci vom efectua doar următoarele acţiuni:

(1) $p=p-1$;

(2) $pmin=min\{pmin-1, -1\}$;

(3) $pmax=max\{pmax-1, 0\}$.

Dacă $S(i)=$*”P”*, vom calcula câte şiruri S' cu proprietăţile date există, astfel încât $S'(j)=S(j)$, pentru $1 \le j \le i-1$, dar $S'(i)=$*”L”*. Fie $nextpmin=min\{pmin-1, -1\}$ şi $nextpmax=max\{pmax-1, -1\}$. Dacă $nextpmin \ge -D$ şi $nextpmax \le D$, atunci vom continua, efectuând următorii paşi:

(1) $nextpmin=min\{0, nextpmin\}$ şi $nextpmax=max\{0, nextpmax\}$;

(2) $nextpmin=-D-nextpmin$ şi $nextpmax=D-nextpmax$;

(3) dacă $-D \le nextpmin \le 0$ şi $0 \le nextpmax \le D$, atunci vom seta $cnt=(cnt+NS(N-i, nextpmin, nextpmax)) \bmod M$.

După aceşti paşi (pentru cazul $S(i)=$*”P”*), va trebui să actualizăm valorile p, $pmin$ şi $pmax$, înainte de a trece la poziţia $i+1$. Vom seta:

(1) $p=p+1$;

(2) $pmin=min\{pmin+1, 0\}$;

(3) $pmax=max\{pmax+1, 1\}$.

Complexitatea algoritmului descris este $O(N \cdot (D+1)^2+N)$.

Problema 7-22. Munte

Linia orizontului dintr-un peisaj se poate desena pe un caroiaj, cu ajutorul caracterelor '/' şi '\', aceasta conturându-se într-un “zig-zag” obişnuit. O linie a orizontului este corectă dacă este desenată cu acelaşi număr de caractere '/', respectiv '\' şi fiecare coloană a caroiajului conţine un singur caracter. De asemenea, această linie trebuie să pornească dintr-o celulă a caroiajului aflată la nivelul mării şi se termină tot într-o celulă aflată la nivelul mării. Pe parcursul trasării, linia nu coboară niciodată sub nivelul mării. Linia orizontului începe cu

caracterul '/' şi se termină cu caracterul '\'. După caracterul '/' poate fi plasat pe coloana următoare caracterul '/' pe linia superioară sau caracterul '\' pe aceeaşi linie. Analog, după caracterul '\' poate fi plasat pe coloana următoare caracterul '\' pe linia inferioară sau caracterul '/' pe aceeaşi linie. Definim vârful muntelui ca fiind format din două caractere /\ situate pe aceeaşi linie şi pe coloane alăturate. Calculaţi numărul de posibilităţi de a trasa linii ale orizontului cu un număr precizat (n) de perechi de caractere '/' şi '\', astfel încât să existe exact k vârfuri.

Exemplu: n=4, k=3 => Răspunsul este 6.

Soluţie: Vom calcula următoarele valori $Num(i,j,q,l)$=numărul de şiruri de caractere formate din i caractere '/', j caractere '\', conţine q vârfuri, iar ultimul caracter al şirului este l (l='/' sau '\'); avem $0 \le j \le i \le n$ şi $0 \le q \le min\{j,k\}$. Vom calcula valorile $Num(i,j,q,l)$ în ordine crescătoare a sumei $(i+j)$. Iniţial avem $Num(1,0,0,'/')=1$ şi $Num(1,0,0,'\backslash')=Num(0,1,*,*)=0$.

Pentru $(i+j) \ge 2$ ($j \le i$) şi orice valoare a lui q ($0 \le q \le j$) avem:

1) $Num(i,j,q,'/')=Num(i-1,j,q,'/')+Num(i-1,j,q,'\backslash')$;

2) $Num(i,j,q,'\backslash')=(Num(i,j-1,q,'\backslash')$ (sau 0, dacă $j=0$)) + ($Num(i,j-1,q-1,'/')$ (sau 0, dacă $j=0$ sau $q=0$)). Rezultatul final îl avem în $Num(n,n,k,'\backslash')$.

Problema 7-23. Numărul de Cuplaje Perfecte într-un Graf Bipartit aproape Complet

Considerăm un graf bipartit ce are N noduri în partea stângă şi N noduri în partea dreaptă. Există muchie (i,j) între oricare nod i ($1 \le i \le N$) din partea stângă şi oricare nod j ($1 \le j \le N$) din partea dreaptă, cu excepţia muchiilor $(1,1), ..., (P,P)$ ($0 \le P \le N$). Determinaţi numărul de cuplaje perfecte ce se pot forma în acest graf.

Soluţie: Vom folosi principiul includerii şi excluderii. Vom genera toate submulţimile mulţimii $\{1, ..., P\}$. Fie S submulţimea curentă şi fie $|S|$ numărul de elemente din S ($0 \le |S| \le P$). Fie $NC(S)$=numărul de cuplaje care conţin muchiile (i,i), cu $i \in S$. Avem $NC(S)=(N-|S|)!$. Răspunsul este egal cu suma valorilor $(-1)^{|S|} \cdot NC(S)$ ($S \subseteq \{1,...,P\}$). Această abordare are complexitatea $O(2^P)$. Observând că toate valorile $NC(S)$ sunt egale pentru toate mulţimile S având acelaşi număr de elemente, răspunsul devine egal cu suma valorilor $(-1)^i \cdot C(P,i) \cdot (N-i)!$ ($0 \le i \le P$). Am notat prin $C(a,b)$=combinări de a elemente luate câte b. Complexitatea acestui algoritm este $O(P)$. La calculul complexităţilor am ignorat factorul ce presupune lucrul cu numere mari, având $O(N)$ cifre. Pentru o analiză corectă, complexităţile menţionate trebuie înmulţite cu $O(N)$.

Problema 7-24. Numere (TIMUS)

Se consideră şirul infinit X format prin concatenarea tuturor numerelor naturale în baza B ($2 \le B \le 10$). De exemplu, primele cifre ale acestui şir pentru $B=10$ sunt: *123456789101112131415161718192021...* Dându-se un şir S (conţinând numai cifre de la 0 la $B-1$), determinaţi la ce poziţie în X apare S pentru prima dată. Poziţiile lui X sunt numerotate începând de la 1. Şirul S apare la o poziţie p în X dacă $X(p+i-1)=S(i)$ ($1 \le i \le len(S)$). Lungimea lui S este cel mult 300.

Soluţie: Vom dori să determinăm cel mai mic număr natural M în interiorul căruia poate începe şirul S, astfel încât acesta să se potrivească peste restul cifrelor din M, cât şi peste cifrele numerelor ce îl urmează pe M în X. Vom iniţializa pe M la S (dacă $S(1)>0$) sau la numărul care are prima cifră 1, iar restul cifrelor sunt cele corespunzătoare lui S (dacă

$S(1)=0$). De asemenea, vom reţine şi poziţia minimă *poz* din cadrul lui *M* la care poate începe şirul *S* (pentru iniţializare vom avea $poz=1$ dacă $S(1)>0$, respectiv $poz=2$, dacă $S(1)=0$).

Vom considera acum două cazuri. În primul caz vom considera că există un număr *Q* complet în interiorul lui *S*. Pentru aceasta, vom considera toate posibilităţile pentru acest număr *Q* (adică toate perechile (i,j), cu $1 \le i \le j \le len(S)$, unde $Q=S(i)S(i+1)...S(j)$). După fixarea acestui număr *Q*, trebuie să verificăm dacă presupunerea făcută este validă. Pentru aceasta vom iniţializa $Q'=Q$ şi $poz'=j$. Cât timp $poz'<len(S)$ vom efectua următorii paşi:

(1) $Q'=Q'+1$;

(2) verificăm dacă *Q'* se potriveşte peste cifrele lui *S*, începând de la poziţia $poz'+1$ şi până la $min\{len(S), poz'+len(Q')\}$;

(3) dacă răspunsul este afirmativ, atunci setăm $poz'=min\{len(S), poz'+len(Q')\}$, altfel oprim iteraţiile ciclului.

Dacă la final am obţinut $poz'=len(S)$, atunci *S* se potriveşte peste numerele ce îi urmează lui *Q*. Vom verifica acum, în mod similar, dacă *S* se potriveşte peste numerele ce îi preced lui *Q*. Iniţializăm $Q'=Q$, $poz'=i$ şi $pozQ=1$. Cât timp $poz'>1$ şi $Q'>1$ efectuăm următorii paşi:

(1) $Q'=Q'-1$;

(2) verificăm dacă *Q'* se potriveşte peste poziţiile lui *S*, începând de la $poz'-1$ şi până la $max\{poz'-len(Q'), 1\}$, efectuând comparaţiile de la ultima cifră a lui *Q'* spre prima (respectiv de la $poz'-1$ descrescător, pentru poziţiile lui *S*);

(3) dacă răspunsul este afirmativ, atunci: dacă $poz'>len(Q')$ atunci $pozQ=1$ altfel $pozQ=len(Q')-poz'+2$; apoi setăm $poz'=max\{poz'-len(Q'), 1\}$ (dacă răspunsul nu e afirmativ, oprim ciclul).

Dacă verificările în ambele sensuri au determinat rezultate valide, atunci: dacă $(Q<M)$ sau ($Q=M$ şi $pozQ<poz$) vom seta $M=Q$ şi $poz=pozQ$.

Al doilea caz consideră că nu există niciun număr *Q* aflat complet în interiorul lui *S*. Pentru acest caz vom considera toate prefixele $Q=S(1)...S(i)$ ($len(S)/2 \le i \le len(S)-1$ şi $S(i+1)>0$). Fie $Q'=Q+1$. Dacă $len(Q')>len(Q)$ (la adunarea cu *1* s-a generat transport), atunci păstrăm în *Q'* doar ultimele $len(Q)$ cifre, şi setăm $t=1$ (altfel lăsăm *Q'* aşa cum e şi setăm $t=0$). Vom încerca acum toate poziţiile *j* ($i+2 \le j \le len(S)+1$) şi vom verifica dacă *Q'* se potriveşte peste cifrele $S(j)S(j+1)...S(len(S))$, adică dacă aceste cifre reprezintă un prefix al lui *Q'* (dacă $j=len(S)+1$, atunci *Q'* se potriveşte în mod implicit). Dacă am obţinut o potrivire pentru o poziţie *j*, atunci fie $Q''=S(i+1)...S(j-1)$. Setăm $Q''=Q''-t$, apoi obţinem *MQ* prin concatenarea lui *Q''* şi a lui *Q* şi setăm $pozQ=len(Q'')+1$. Dacă ($Q''>0$) şi (($MQ<M$) sau ($MQ=M$ şi $pozQ<poz$)) atunci setăm $M=MQ$ şi $poz=pozQ$.

După considerarea tuturor acestor cazuri am găsit cel mai mic număr *M* în interiorul căruia poate începe numărul *S* (şi poziţia minimă *poz* la care începe). Mai trebuie doar să determinăm câte cifre au în total numerele de la *1* până la *M-1*. Fie $ndigits(M-1)$ acest număr. Rezultatul final va fi $ndigits(M-1)+poz$.

Pentru a calcula $ndigits(Q)$ (pentru un număr oarecare *Q*), vom proceda după cum urmează. Dacă $Q=0$, rezultatul este *0*. Altfel, iniţializăm $rez=0$. Pentru $i=1,...,len(Q)-1$ incrementăm *rez* cu $i \cdot (B-1) \cdot B^{i-1}$ (există $(B-1) \cdot B^{i-1}$ numere de *i* cifre, care nu încep cu cifra *0*). Apoi mai trebuie să adunăm la *rez* suma cifrelor tuturor numerelor de $len(Q)$ cifre mai mici sau egale cu *Q*. Vom parcurge, pe rând, cifrele numărului *Q* (cu *i* de la *1* la $len(Q)$). Pentru $i=1$ adunăm la *rez* valoarea $len(Q) \cdot (Q(i)-1) \cdot B^{len(Q)-i}$. Pentru $i>1$, adunăm la *rez* valoarea $len(Q) \cdot Q(i) \cdot B^{len(Q)-i}$. La final mai adăugăm la *rez* valoarea $len(Q)$ (pentru a aduna şi cifrele numărului *Q*). *rez* va reprezenta rezultatul întors de $ndigits(Q)$.

Trebuie menţionat că o implementare a acestei probleme trebuie să lucreze cu numere mari (de sute de cifre). După fiecare operaţie, rezultatul trebuie să fie un număr valid (adică fără *0*-uri la început). De asemenea, toate operaţiile trebuie să se efectueze în baza *B*.

Problema 7-25. Borg (ONI 2006)

Oricine a urmărit serialul Star Trek îşi aduce aminte de borgi şi de nava lor spaţială în formă de cub. Una dintre problemele pe care şi-au pus-o înainte de a construi nava a fost următoarea. Nava borgilor are forma unui paralelipiped dreptunghic de dimensiuni *NxMxH*, împărţit în camere de dimensiune *1x1x1*. Pentru ca nava să poată funcţiona, în aceste camere trebuie plasate *K* motoare de propulsie, în fiecare cameră putându-se plasa cel mult un motor. O cameră poate fi identificată printr-un triplet *(a, b, c)*, unde $1 \le a \le N$, $1 \le b \le M$, $1 \le c \le H$, reprezentând coordonatele sale. Un plan al paralelipipedului este o mulţime de camere de unul dintre următoarele *3* tipuri:

{(a, b, c) | a fixat, 1 ≤ b ≤ M, 1 ≤ c ≤ H} – în total sunt *N* plane de acest tip;
{(a, b, c) | b fixat, 1 ≤ a ≤ N, 1 ≤ c ≤ H} – în total sunt *M* plane de acest tip;
{(a, b, c) | c fixat, 1 ≤ a ≤ N, 1 ≤ b ≤ M} – în total sunt *H* plane de acest tip.

Se cere să se găsească *R*, numărul de moduri (*modulo 30103*) în care se pot plasa cele *K* motoare, astfel încât orice plan al paralelipipedului să conţină cel puţin o cameră ocupată de un motor.

Soluţie: Numărul total de moduri în care se pot plasa *K* motoare în paralelipiped, fără restricţia ca fiecare plan să conţină cel puţin un motor, este C_{N*M*H}^{K}. Din acestea, trebuie să le scădem pe cele care nu sunt bune. Dacă notăm cu X_i (*1≤i≤N*) numărul de moduri în care se pot plasa cele *K* motoare astfel încât planul *i* (de tip *1*) să fie gol, cu Y_i (*1≤i≤M*) numărul de moduri astfel încât planul *i* (de tip *2*) să fie gol şi cu Z_i (*1≤i≤ H*) numărul de moduri astfel încât planul *i* (de tip *3*) să fie gol, atunci numărul căutat va fi $C_{N*M*H}^{K} - \left| \bigcup_{1\le i\le N} X_i \cup \bigcup_{1\le i\le M} Y_i \cup \bigcup_{1\le i\le H} Z_i \right|$. Cardinalul reuniunii se poate dezvolta cu principiul includerii şi excluderii, fiecare mulţime rezultată conţinând o intersecţie de *a* mulţimi *X*, *b* mulţimi *Y* şi *c* mulţimi *Z*. Cardinalul unei astfel de mulţimi ar reprezenta numărul de moduri în care se pot pune *K* motoare, astfel încât *a* dintre planele de tip *1* să fie goale, *b* dintre planele de tip *2* să fie goale şi *c* dintre planele de tip *3* să fie goale, adică ar fi $C_{(N-a)*(M-b)*(H-c)}^{K}$. În dezvoltarea produsă de principiul includerii şi excluderii vor fi $C_N^a * C_M^b * C_H^c$ astfel de mulţimi.

Aşadar numărul total va fi $\sum_{a=1}^{N}\sum_{b=1}^{M}\sum_{c=1}^{H}(-1)^{a+b+c} * C_N^a * C_M^b * C_H^c * C_{(N-a)*(M-b)*(H-c)}^{K}$.

Se observă că este nevoie de toate combinările până la maximul dintre *N*, *M* şi *H*, dar numai de cele care au *K* sus pentru restul. Acestea se pot calcula în complexitatea *O(N·M·H·K)* folosind regula generală $C_n^k = C_{n-1}^{k-1} + C_{n-1}^{k}$ (*k≤K şi n≤N·M·H*). Pentru a reduce complexitatea la *O(N·M·H)*, putem calcula C_n^K direct din C_{n-1}^K, folosindu-ne de faptul ca

numărul *30103* este prim şi deci orice număr are un invers modular (care înmulţit cu el dă restul *1* la împărţirea la *30103*).

Capitolul 8. Teoria Jocurilor

Problema 8-1. Jocul cu pietre (TIMUS)

Avem la dispoziţie N ($1 \le N \le 100.000$) grămezi. Fiecare grămadă i ($1 \le i \le N$) conţine $P(i)$ pietre ($0 \le i \le 10^{1000}$). Doi jucători efectuează mutări alternativ. O mutare constă în alegerea unei grămezi i şi eliminarea a X pietre din grămadă, unde X este o putere a lui 2 ($1, 2, 4, 8, 16, ...$), cu condiţia ca grămada să mai conţină cel puţin X pietre. Jucătorul care nu mai poate efectua nicio mutare atunci când îi vine rândul pierde jocul. Determinaţi cine câştigă (jucătorul 1 fiind cel care efectuează prima mutare), considerând că ambii jucători joacă optim.

Soluţie: Un joc este echivalent cu o grămadă de pietre. O stare a unui joc simplu este determinată prin numărul de pietre din cadrul grămezii. Vom încerca să calculăm $G(Q)$=numărul Grundy asociat unei grămezi ce conţine Q pietre.

Avem $G(0)=0$. Pentru $Q \ge 1$, putem ajunge în orice stare $Q' \ge 0$, cu proprietatea că $Q-Q'=2^i$ ($i \ge 0$). Aplicând teoria numerelor Sprague-Grundy, obţinem $G(1)=1$, $G(2)=2$, $G(3)=0$, $G(4)=1$, $G(5)=2$, $G(6)=0$, $G(7)=1$, $G(8)=2$, ş.a.m.d. Observăm uşor un tipar: $G(Q)=Q \text{ mod } 3$ (restul împărţirii lui Q la 3). Această regulă poate fi demonstrată prin inducţie. Dintr-o stare Q putem ajunge numai în stări Q' astfel încât $(Q \text{ mod } 3) \ne (Q' \text{ mod } 3)$. Pentru $Q \ge 2$ putem ajunge mereu în stări cu resturi din mulţimea $\{0,1,2\} \setminus \{Q \text{ mod } 3\}$ (de exemplu, stările $Q-1$ şi $Q-2$ sunt stările cu 2 resturi diferite de ($Q \text{ mod } 3$)). Presupunând că regula este adevărată până la un număr de pietre R, atunci, folosind această observaţie, ea va fi adevărată şi pentru $R+1$.

Pentru jocul compus calculăm valoarea $Z=G(P(1)) \text{ xor } G(P(2)) \text{ xor } ... \text{ xor } G(P(N))$ (unde, după cum am arătat deja, $G(P(i))=P(i) \text{ mod } 3$). Dacă $Z>0$, primul jucător va câştiga jocul; altfel, al doilea jucător are strategie sigură de câştig.

Problema 8-2. DoiPatru (Lotul Naţional de Informatică, România 2002)

Membrii Lotului Naţional de Informatică sunt foarte mândri de noul joc inventat de ei, pe care l-au denumit asemănător cu o problemă de la Olimpiada Internaţională de Informatică din 2001, care le-a plăcut foarte mult. Astfel, jocul se numeşte **DoiPatru**. Pentru acest joc se folosesc N ($1 \le N \le 30$) grămezi, fiecare conţinând **cel puţin 0** şi **cel mult 4** bile. Numărul total de bile din toate grămezile este $2 \cdot N$. Doi jucători mută alternativ. Atunci când îi vine rândul, fiecare jucător este obligat să efectueze o mutare validă. O mutare validă constă din alegerea a două grămezi, dintre care prima grămadă are mai multe bile decât cea de a doua. Jucătorul ia o bilă din prima grămadă şi o mută în cealaltă. Mutarea se consideră validă, doar dacă numărul de bile rezultat în a doua grămadă după mutarea bilei nu este mai mare decât numărul de bile rămas în prima grămadă. Jocul se termină atunci când nu mai poate fi efectuată nici o mutare validă (dacă vă gândiţi puţin, veţi constata că acest lucru se întâmplă atunci când fiecare grămadă conţine două bile).

Câştigătorul jocului este desemnat cel care deţine mai multe grămezi la sfârşitul jocului. Bineînţeles, dacă cei doi jucători deţin un număr egal de grămezi, jocul se consideră a fi remiză.

Un jucător deţine o grămadă dacă grămada are două bile, iar acest număr (de două bile) a rezultat în urma unei mutări efectuate de jucătorul respectiv. De exemplu, dacă un jucător

alege o grămadă cu **4** bile şi una cu o bilă, în urma efectuării mutării, el va deţine cea de-a doua grămadă (care va avea două bile), dar prima nu va aparţine deocamdată nici unuia dintre jucători. Dacă alege o grămadă cu **3** bile şi una cu **0** bile, jucătorul va deveni proprietarul primei grămezi, deoarece, în urma mutării efectuate, grămada respectivă va rămâne cu două bile. În cazul în care alege o grămadă cu **3** bile şi una cu o bilă, după efectuarea mutării, el va deţine ambele grămezi (amândouă au acum două bile).

Dacă un jucător este proprietarul unei grămezi la un moment dat în timpul jocului, nu înseamnă că această grămadă va rămâne în posesia lui până la sfârşit. De exemplu, să presupunem că jucătorul **1** deţine o grămadă cu două bile şi este rândul jucătorului **2** să mute. Dacă acesta alege o grămadă cu **4** bile şi grămada cu două bile ce aparţine jucătorului **1**, după efectuarea mutării, ambele grămezi vor avea **3** bile, iar numărul de grămezi aflate în posesia jucătorului **1** va scădea cu **1** (grămada deţinută de el anterior nu mai aparţine nici unuia din cei doi jucători, căci nu mai are două bile).

Dacă la începutul jocului există unele grămezi având două bile, acestea sunt distribuite în mod egal celor doi jucători. Dacă numărul de grămezi cu două bile este impar, atunci jucătorul **2** va primi cu o grămadă mai mult decât jucătorul **1**. Jucătorul **1** este cel care efectuează prima mutare. Pentru un **N** dat şi un set de configuraţii iniţiale ale jocului cu **N** grămezi, decideţi rezultatul fiecărei configuraţii de joc (considerând că ambii jucători joacă optim).

Exemple:
N=5
0 3 4 1 2 => Câştigă primul jucător.
2 2 2 2 2 => Câştigă cel de-al doilea jucător.
1 1 2 2 4 => Câştigă primul jucător.
4 3 2 1 0 => Câştigă primul jucător.

Soluţie: O stare a jocului este o pereche *(C, k)*, unde *C* este o configuraţie a jocului (un şir de *N* numere între *0* şi *4*, sortate crescător, deoarece ordinea lor nu este importantă, a căror sumă este *2·N*), iar *k* reprezintă numărul de grămezi cu *2* bile deţinute de jucătorul aflat la mutare (observăm că nu contează care grămezi cu *2* bile sunt deţinute de jucătorul aflat la mutare şi care de adversar; putem presupune că primele *k* grămezi cu *2* bile din configuraţia *C* aparţin jucătorului aflat la mutare, iar celelalte grămezi cu *2* bile aparţin adversarului). Pentru *N=30* există aproximativ *1000* de configuraţii *C*, iar *k* poate lua valori între *0* şi *N*. Astfel, graful stărilor conţine în jur de *30.000* de noduri.

Pentru fiecare stare *S=(C,t)* vom calcula *bestRes(S)*, conform algoritmului descris la începutul capitolului. Vom încerca, pe rând, fiecare mutare posibilă şi, dacă mutarea poate fi efectuată, vom genera starea *S'* în care ajungem. Dacă nu am calculat încă *bestRes(S')*, apelăm recursiv funcţia de calcul pentru *S'* (aceasta este o versiune recursivă a algoritmului descris la începutul capitolului). Dacă dintr-o stare *S=(C,t)* nu se poate ajunge în nicio altă stare *S'*, atunci *C* conţine numai grămezi cu câte *2* bile. Dacă *N* e par şi *k>N/2*, atunci *bestRes[S]=victorie*; dacă *k=N/2*, atunci *bestRes[S]=remiză*; dacă *k<N/2*, *bestRes[S]=înfrângere*. Pentru *N* impar, dacă *k>N/2*, atunci *bestRes[S]=victorie*, altfel *bestRes[S]=înfrângere*. Pentru celelalte stări *S*, după ce am calculat *bestRes[S']* pentru toate stările *S'* în care putem ajunge din *S*, vom folosi regulile din algoritmul descris la începutul capitolului.

Problema 8-3. Maxi-Nim (TIMUS)

Se dau N ($1 \le N \le 100.000$) grămezi cu pietre. Fiecare grămadă i ($1 \le i \le N$) conţine $P(i)$ pietre ($0 \le P(i) \le 2.000.000.000$). Doi jucători efectuează mutări alternativ. O mutare constă în alegerea unei grămezi j cu proprietatea că $P(j) \ge P(i)$ (oricare ar fi $1 \le i \le N$) (adică alegerea uneia dintre grămezile care conţin un număr maxim de pietre) şi extragerea a unui număr X de pietre (X poate fi orice număr între 1 şi $P(j)$). Cel care extrage ultima piatră câştigă. Determinaţi ce jucător va câştiga, în condiţiile în care ambii vor juca optim (jucătorul 1 este cel care începe jocul).

Soluţie: Să presupunem că Z este numărul maxim de pietre din orice grămadă ($Z = max\{P(i) | 1 \le i \le N\}$). Fie NZ numărul de grămezi i astfel încât $P(i)=Z$. Dacă $NZ \ mod \ 2=1$, atunci jucătorul 1 are strategie sigură de câştig. Altfel, jucătorul 2 are strategie sigură de câştig. Demonstraţia este simplă. Cât timp $NZ>1$ jucătorul 1 poate lua orice număr de pietre din orice grămadă cu Z pietre. Când $NZ=1$ va fi rândul jucătorului 1 să mute (deoarece numărul iniţial de grămezi cu Z pietre era impar). Dacă grămada cu Z pietre este singura grămadă cu număr nenul de pietre, atunci el va lua toate pietrele şi va câştiga jocul. Altfel, fie Z' al doilea număr maxim de pietre dintr-o grămadă ($Z' = max\{P(i) | 1 \le i \le N, P(i)<Z\}$) şi fie NZ' numărul de grămezi cu Z' pietre. Dacă NZ' este impar, atunci el va lua din grămada cu Z pietre ($Z-Z'$) pietre, reducând-o la o grămadă cu Z' pietre şi lăsându-şi adversarul într-o stare în care numărul de grămezi cu număr maxim de pietre este par. Dacă NZ' este par, atunci el va reduce grămada cu Z pietre la o grămadă cu $Z''<Z'$ pietre (luând $Z'-Z''$ pietre din ea). Dacă jucătorul 1 are la început în faţă un număr par de grămezi cu număr maxim Z de pietre, atunci după prima mutare jucătorul 2 se va afla în situaţia în care are un număr impar de grămezi cu număr maxim de pietre şi va putea aplica staregia descrisă mai sus.

Din punct de vedere al implementării, se poate menţine un hash (sau arbore echilibrat) cu perechi *(cheie=Z, valoare=număr de grămezi cu număr de pietre egal cu Z)* şi un heap cu perechi *(P(i), i)*. Elementele din heap sunt ordonate după $P(i)$. Astfel, se poate afla imediat numărul maxim de pietre dintr-o grămadă, precum şi numărul de grămezi care au acest număr. Atunci când se iau X pietre dintr-o grămadă i, se şterge din heap perechea $(P(i), i)$ şi se decrementează în hash numărul de grămezi cu $P(i)$ pietre (dacă acest număr ajunge la 0, atunci perechea respectivă se şterge din hash). Apoi se setază $P(i)=P(i)-X$ şi, dacă $P(i)>0$, se inserează în heap perechea $(P(i), i)$. În hash se incrementează cu 1 numărul de grămezi cu $P(i)$ pietre (dacă acest număr era 0, atunci se inserează perechea $(P(i), 1)$). Atunci când jucătorul aflat la mutare rămâne cu o singură grămadă cu număr maxim Z de pietre, acesta trebuie să verifice dacă este ultima grămadă – verifică dacă este singurul element din heap. Dacă nu este ultima grămadă, el trebuie să determine numărul maxim de pietre Z'<Z. Pentru aceasta se poate uita la cei (maxim) 2 fii ai nodului rădăcină din max-heap. Al doilea maxim se află într-unul din aceşti doi fii. Apoi caută perechea (Z', NZ') în hash, pentru a determina numărul de grămezi NZ' care au Z' pietre.

Problema 8-4. Leftist Nim (TIMUS)

Se dau N ($1 \le N \le 100.000$) grămezi cu pietre, aşezate în ordine crescătoare de la stânga la dreapta. Fiecare grămadă i ($1 \le i \le N$) conţine $P(i)$ pietre ($1 \le P(i) \le 2.000.000.000$). Doi jucători efectuează mutări alternativ. Fie j grămada cu indice cel mai mic care conţine $P(j)>0$ pietre. O mutare constă în extragerea a unui număr X de pietre (X poate fi orice număr între 1 şi $P(j)$) din grămada j. Cel care extrage ultima piatră câştigă. Determinaţi ce jucător va câştiga, în condiţiile în care ambii vor juca optim (jucătorul 1 este cel care începe jocul).

Soluţie: Vom calcula *win(j)=1*, dacă jucătorul aflat la mutare are strategie sigură de câştig, în cazul în care jocul constă doar din grămezile *j, j+1, ..., N* (şi *0*, altfel). Avem *win(N)=1* (deoarece jucătorul aflat la mutare poate lua toate pietrele din grămada *N*). Pentru *1≤j≤N-1* (în ordinea descrescătoare) avem: dacă *P(j)=1* atunci *win(j)=1-win(j+1)*; altfel (dacă *P(j)>1*), *win(j)=1*.

Jocul se va desfăşura astfel. Să presupunem că jucătorul care are strategie sigură de câştig este primul care urmează să ia pietre din grămada *j*, cu *win(j)=1* (*1≤j≤N-1*). Dacă *win(j+1)=0*, atunci el va lua toate cele *P(j)* pietre. Dacă *win(j+1)=1*, atunci el va lua *P(j)-1* pietre; astfel, la următoarea mutare, adversarul va lua ultima piatră din grămada *j*, iar jucătorul *1* va fi primul jucător care va lua pietre din grămada *j+1* (care are *win(j+1)=1*).

Problema 8-5. Vot public (UVA)

În societatea indienilor Uxuhul, votul asupra chestiunilor celor mai importante din societate se desfăşura după cum urmează. Consiliul celor mai înţelepţi indieni se adunau şi cei *N* (*1≤N≤10.000*) membri ai săi se aranjau în linie, în ordine crescătoare a înţelepciunii (cel mai înţelept membru al consilului era ultimul). Apoi, în această ordine, membrii consilului îşi exprimau votul, în felul următor. Votul poate avea *M* (*1≤M≤50*) stări. Iniţial, starea votului este *Q* (*1≤Q≤M*). Când vine rândul unui membru *i*, acesta poate schimba starea curentă a votului. Pentru fiecare membru *i* (*1≤i≤N*) şi fiecare stare curentă posibilă *s* (*1≤s≤M*), se cunoaşte lista *L(i,s)* a stărilor în care poate fi modificată starea curentă *s* (*L(i,s)* va conţine cel puţin un element; *s* poate face parte din *L(i,s)*).

După ce votează şi ultimul membru, rezultatul final al votului este starea în care a fost adus votul de ultimul membru. Fiecare membru *i* are, de asemenea, o preferinţă (distinctă) pentru fiecare stare posibilă *s*, *Pref(i,s)*. Mai exact, dacă *Pref(i,sa)>Pref(i,sb)*, atunci el ar dori ca rezultatul final al votului să fie *sa*, mai degrabă decât *sb*. Atunci când votează, fiecare membru modifică starea curentă a votului în aşa fel încât rezultatul final să aibă o preferinţă cât mai mare pentru el. Determinaţi rezultatul final al votului.

Soluţie: Vom calcula un tabel *Rez(i,s)*=rezultatul final al votului, dacă atunci când îi vine rândul membrului *i* să voteze, votul se află în starea *s*. Avem *Rez(N+1,s)=s*. Pentru *1≤i≤N* (în ordine descrescătoare) şi orice stare *s* (*1≤s≤M*), vom calcula *Rez(i,s)* după cum urmează. Vom considera toate stările *s'* din *L(i,s)* şi vom evalua rezultatul *r'=Rez(i+1,s')*, dacă membrul *i* schimbă starea votului în *s'*. Fie *Rezset(i,s)*=mulţimea rezultatelor *r'* ce pot fi obţinute astfel. *Rez(i,s)=rmax*, cu proprietatea că *Pref(i,rmax)>Pref(i,r'')* (cu *rmax* din *Rezset(i,s)* şi oricare ar fi *r''≠rmax* făcând parte tot din *Rezset(i,s)*).

Rez(1,Q) reprezintă rezultatul final al votului. Complexitatea algoritmului este *O(N·M+Msum)*, unde *Msum* este suma cardinalelor mulţimilor *L(i,s)* (*1≤i≤N*; *1≤s≤M*). *Msum* poate fi de ordinul $O(N{\cdot}M^2)$, dar, pentru anumite situaţii speciale (de ex., *L(i,s)* conţine doar un număr de elemente limitat de o constantă, indiferent de valoarea lui *M*), *Msum* poate fi mai mic.

Problema 8-6. Pietre pe o tablă infinită (TIMUS)

Se dă o tablă bidimensională infinită. Undeva pe această tablă se află *M·N* pietre, aşezate într-un dreptunghi cu *M* linii şi *N* coloane (*1≤M,N≤1.000*). Se pot efectua mutări de tipul următor: se alege o piatră *A* şi se sare cu ea peste o piatră vecină *B* (pe orizontală sau verticală). După săritură, piatra *A* îşi schimbă poziţia; noua poziţie trebuie să fie liberă

înaintea săriturii; de asemenea, în urma săriturii, piatra *B* este eliminată de pe tablă. Dacă o piatră nu are nicio piatră vecină pe orizontală sau verticală, atunci ea nu poate sări. Determinaţi numărul minim de pietre care mai pot rămâne pe tablă, în urma efectuării unei secvenţe de mutări corespunzătoare.
Exemplu: M=3, N=4 => Răspunsul este 2.

Soluţie: În mod evident, nu contează care din cele două numere (*M* sau *N*) reprezintă liniile sau coloanele. Vom considera că $M<N$. Dacă $M=1$, atunci toate pietrele sunt aşezate în linie. În acest caz, să presupunem pietrele numerotate de la *1* la *N* (de la stânga la dreapta). Vom efectua următoarele mutări: piatra *2* sare peste piatra *1*, piatra *4* peste piatra *3*, ..., piatra $2\cdot i$ peste piatra $2\cdot i-1$ ($1\le i\le N/2$). La final, vor rămâne *((N+1) div 2)* pietre.

Dacă $M\geq 2$ (implicit şi $N\geq 2$), atunci avem *2* cazuri: dacă *(M mod 3=0)* sau *(N mod 3=0)*, atunci răspunsul este *2*; altfel, răspunsul este *1*. Demonstraţia acestor cazuri este lăsată drept exerciţiu cititorului (găsiţi o strategie care garantează că rămân *2*, respectiv *1* piatră la sfârşit; pentru cazul în care rămân *2* pietre, arătaţi că nu există o altă strategie în urma căreia să rămână o singură piatră).

Capitolul 9. Probleme cu Matrici

Problema 9-1. Staţii radio (TIMUS)

Se dă o matrice H având M linii şi N coloane ($1 \le M,N \le 100$). Fiecare element $H(i,j)$ al matricii reprezintă o altitudine ($0 \le H(i,j) \le 30.000$; $H(i,j)$ este un număr întreg). În K ($1 \le K \le 1.000$) pătrăţele din matrice se află amplasată câte o staţie radio. Staţia i ($1 \le i \le K$) se află amplasată pe linia $L(i)$ ($1 \le L(i) \le M$), coloana $C(i)$ ($1 \le C(i) \le N$) şi are o rază de transmisie $R(i)$ ($R(i)$ poate fi un număr real). Dorim să amplasăm un receptor radio într-unul din pătrăţelele în care nu se află amplasată nicio staţie radio. Receptorul poate fi amplasat la înălţimea pătrăţelului respectiv sau la orice înălţime mai mare, care să fie, însă, număr întreg. Receptorul radio trebuie să recepţioneze semnalele de la toate cele K staţii radio (adică să se afle amplasat în raza lor de transmisie). Determinaţi câte modalităţi diferite există pentru amplasarea receptorului radio (două modalităţi se consideră diferite şi dacă receptorul este amplasat în acelaşi pătrăţel, dar la înalţimi diferite). Se consideră că semnalul unei staţii radio i poate fi recepţionat oriunde în sfera de rază $R(i)$ cu centrul la $(L(i), C(i), H(L(i), C(i)))$; se foloseşte distanţa euclidiană (norma L_2).

Soluţie: Vom considera fiecare pătrăţel (i,j) din matrice şi vom calcula $Npos(i,j)$=numărul posibilităţilor de amplasarea a receptorului în pătrăţelul (i,j). Dacă există deja o staţie radio amplasată în pătrăţelul (i,j), atunci $Npos(i,j)=0$. Altfel, pentru fiecare staţie radio q ($1 \le q \le K$) vom calcula un interval $[hmin(q), hmax(q)]$ în care poate fi amplasat receptorul pentru a recepţiona semnalul staţiei i. Calculăm întâi distanţa în planul XOY:

$dxoy(q)=sqrt((i-L(q))^2+(j-C(q))^2)$.

Dacă $dxoy(q)>R(q)$, atunci setăm $hmin(q)=0$ şi $hmax(q)=hmin(q)-1$. Altfel, vom calcula $dH=sqrt(R(q)^2-dxoy^2)$.

Receptorul poate fi amplasat oriunde în intervalul $[hmin(q)=H(L(q),C(q))-dH, H(L(q),C(q))+dH]$ deasupra pătrăţelului (i,j). După calcularea tururor acestor intervale vom mai adăuga şi intervalul $[hmin(K+1)=H(i,j), hmax(K+1)=+\infty]$ şi vom calcula intersecţia celor $K+1$ intervale: $hminint=max\{hmin(q) | 1 \le q \le K+1\}$, $hmaxint=min\{hmax(q) | 1 \le q \le K+1\}$. Vom rotunji apoi $hminint$ la cel mai apropiat întreg $hmi \ge hminint$ şi $hmaxint$ la cel mai apropiat întreg $hma \le hmaxint$. Dacă $hmi>hma$, atunci $Npos(i,j)=0$; altfel, $Npos(i,j)=hma-hmi+1$. Răspunsul este suma valorilor $Npos(*,*)$. Complexitatea algoritmului este $O(M \cdot N \cdot K)$.

Problema 9-2. Matrice binară cu sume date pe linii şi pe coloane

Se doreşte construcţia unei matrici binare (cu elemente 0 sau 1) cu M linii şi N coloane ($1 \le M,N \le 1.000$), astfel încât suma elementelor de pe fiecare linie i să aparţină intervalului $[Slinmin(i), Slinmax(i)]$ şi suma elementelor de pe fiecare coloană j să aparţină intervalului $[Scolmin(j), Scolmax(j)]$. Valorile unor elemente (i,j) sunt fixate la 0 sau la 1.

Soluţie: Vom începe prin a decrementa cu 1 valorile $Slinmin(i)$, $Slinmax(i)$, $Scolmin(j)$ şi $Scolmax(j)$, pentru fiecare poziţie (i,j) care este fixată la valoarea 1. Vom construi apoi o reţea de flux, după cum urmează. Vom considera nodurile $l(1), ..., l(M)$ şi $c(1), ..., c(N)$. Vom avea o muchie de capacitate superioară 1 şi capacitate inferioară 0 între $l(i)$ şi $c(j)$, dacă elementul (i,j) nu este fixat la nicio valoarea (fie ea 0 sau 1). Vom adăuga apoi alte două noduri, S şi D. Vom adăuga muchii de la S la fiecare nod $l(i)$, care au capacitate inferioară

Slinmin(i) şi capacitate superioară *Slinmax(i)*. Vom duce, de asemenea, muchii de la fiecare nod *c(j)* la nodul *D*, de capacitate inferioară *Scolmin(j)* şi capacitate superioară *Scolmax(j)*. Vom încerca apoi să găsim un flux valid, care respectă restricţiile inferioare şi superioare de pe fiecare muchie. Vom folosi algoritmul descris în [CLRS], care se reduce la determinarea unui flux maxim într-o reţea de flux modificată. Putem implementa algoritmul de flux maxim în complexitate $O((M+N)^3)$. Apoi, dacă avem flux *1* pe o muchie *(l(i), c(j))*, atunci elementul *(i,j)* din matrice va avea valoarea *1* (altfel, *(i,j)* va avea valoarea *0*).

Dacă *Slinmin(i)=Slinmax(i)* şi *Scolmin(j)=Scolmax(j)* pentru orice linie *i* şi orice coloană *j*, atunci putem calcula direct doar un flux maxim. Vom construi aceeasi reţea de flux, în care toate capacităţile inferioare vor fi *0*. Dacă fluxul maxim în acea reţea (considerând doar capacităţi superioare) este egal cu suma valorilor *Slinmax(i)* ($1 \le i \le M$) (această sumă trebuind să fie egală şi cu suma valorilor *Scolmax(j)*, $1 \le j \le N$), atunci există o soluţie. Din nou, elementele *(i,j)* cu valoarea *1* din matrice sunt acelea pentru care fluxul de pe muchia *(l(i), c(j))* este *1* (celelalte elemente nefixate în prealabil vor avea valoarea *0*).

Pentru cazul în care *Slinmin(i)=Slinmax(i)* şi *Scolmin(j)=Scolmax(j)* pentru orice linie *i* şi orice coloană *j*, precum şi dacă nu este fixat niciun element în prealabil, putem obţine un algoritm cu complexitatea $O(M \cdot N)$. Vom nota *Slin(i)=Slinmax(i)* şi *Scol(j)=Scolmax(j)*. Vom sorta liniile descrescător după *Slin(*)* şi coloanele descrescător după *Scol(*)*. Astfel, vom avea $Slin(i) \ge Slin(i+1)$ ($1 \le i \le M-1$) şi $Scol(j) \ge Scol(j+1)$ ($1 \le j \le N-1$). Vom parcurge liniile în ordinea sortată, de la prima către ultima. Pentru a amplasa cele *Slin(i)* elemente egale cu *1* pe linia *i*, vom alege primele *Slin(i)* coloane (în ordinea sortată a coloanelor). Apoi vom decrementa cu *1* valorile *Scol(*)* ale primelor *Slin(i)* coloane. După aceasta, vom resorta valorile *Scol(*)* corespunzătoare coloanelor. Resortarea se poate realiza în timp *O(N)*, deoarece constă în interclasarea primelor *Slin(i)* valori (a căror ordine relativă se menţine) cu celelalte *N-Slin(i)* valori (care nu au fost decrementate cu *1*). Astfel, obţinem complexitatea dorită.

Bineînţeles, atunci când sortăm liniile şi (re)sortăm coloanele, vom menţine, pentru fiecare valoare *Slin(i)*, respectiv *Scol(j)*, coloana căreia îi corespunde. Astfel, pentru fiecare linie *i*, cele *Slin(i)* elemente egale cu *1* vor fi amplasate pe coloanele corespunzătoare primelor (celor mai mari) *Slin(i)* valori *Scol(*)*.

Capitolul 10. Probabilităţi

Problema 10-1. Bile într-o urnă (Olimpiada Online 2001, România)

Într-o urnă se află *W* ($1 \le W \le 100$) bile albe şi *B* ($1 \le B \le 100$) bile negre. Doi jucători extrag, pe rând, câte o bilă şi se uită la ea. Daca e albă o păstrează, altfel (dacă e neagră) o pune la loc. Câştigă cel care extrage ultima bilă albă. În total, pot avea loc maxim *K* extrageri. Dacă nimeni nu a câştigat după *K* ($1 \le K \le 100$) extrageri, atunci jocul se consideră remiză. Determinaţi probabilitatea ca primul jucător (cel care efectuează prima extragere) să câştige jocul.

Soluţie: Vom calcula două matrici: $PA(q,a)$=probabilitatea ca primul jucător să câştige jocul dacă el se află la mutare, au mai rămas *a* bile albe şi mai sunt posibile, în total, *q* extrageri, şi $PB(q,a)$=probabilitatea ca primul jucător să câştige jocul dacă la mutare se află al doilea jucător, au mai rămas *a* bile albe şi mai sunt posibile, în total, *q* extrageri.

Avem $PA(0,*)=PB(0,*)=0$. Pentru $1 \le q \le K$ avem următoarele relaţii:

(1) $PA(q,0)=PB(q,0)=0$;

(2) $PA(q,1)=(1/(B+1))+(B/(B+1))\cdot PB(q-1,1)$;

(3) $PB(q,1)=(1/(B+1))\cdot PA(q-1,0)+(B/(B+1))\cdot PA(q-1,1);$

(4) $PA(q,2 \le a \le W)=(a/(a+B))\cdot PB(q-1,a-1)+(b/(a+b))\cdot PB(q-1,a);$

(5) $PB(q,2 \le a \le W)=(a/(a+B))\cdot PA(q-1,a-1)+(b/(a+b))\cdot PB(q-1,a).$

Relaţiile se bazează pe următoarele fapte. Atunci când există *a* bile albe şi *b* bile negre în urnă, există o probabilitate de $a/(a+b)$ să fie extrasă o bilă albă şi una de $b/(a+b)$ să fie extrasă o bilă neagră. Dacă se extrage o bilă albă, numărul de bile albe scade cu *1*; altfel rămâne constant. Indiferent de ce jucător se află la mutare, la mutarea următoarea va efectua o extragere celălalt jucător. Rezultatul problemei este $PA(K,W)$, iar complexitatea algoritmului este evidentă: $O(W\cdot K)$.

Problema 10-2. Joc de cărţi (Olimpiada Judeţeană de Informatică, Bucureşti, România, 2001)

Într-un pachet de cărţi se află *N* cărţi nerge şi *R* cărţi roşii ($1 \le N,R \le 50.000$), aşezate într-o ordine oarecare în pachet, cu faţa în jos (orice ordine este la fel de probabilă). Un jucător joacă următorul joc. El încearcă să ghicească culoarea cărţii aflate în vâful pachetului.

Pentru aceasta, el alege o culoare (roşu sau negru) şi întoarce cartea din vârful pachetului. Dacă a ghicit culoarea, el elimină cartea din pachet şi continuă cu încercarea de a ghici culoarea următoarei cărţi. Dacă nu a ghicit culoarea, jocul se termină. Jocul se termină şi dacă a ghicit culorile tuturor cărţilor din pachet. Jucătorul nu ghiceşte culorile cărţilor, însă, la întâmplare. El ştie, la orice moment de timp, numărul de cărţi nerge (*Ncur*) şi roşii (*Rcur*) care au mai răms în pachet. Dacă $Rcur \ge Ncur$, el alege culoarea roşie; altfel, alege culoarea neagră. Determinaţi numărul mediu de cărţi ale căror culori jucătorul le ghiceşte până la sfârşitul unui joc.

Exemplu: N=1, R=1 => Răspunsul este 1.0.

Soluţie: Răspunsul este egal cu suma valorilor $(L\cdot Prob(L))$ ($0 \le L \le N+R$), unde $Prob(L)$ este probabilitatea ca jucătorul să ghicească exact *L* cărţi la un joc. $Prob(L)$ poate fi definit ca $Nconfig(L)/Ntotal(N+R)$, unde $Ntotal(N+R)$ este numărul total de configuraţii conţinând *N*

cărţi nerge şi R cărţi roşii (care este egal cu $C(N+R,N)$=combinări de $N+R$ luate câte N), iar $Nconfig(L)$=numărul de configuraţii în care se ghicesc exact L cărţi. Observăm că dacă s-au ghicit exact L cărţi, ştim exact ordinea primelor L cărţi. Să definim $Ncur(L)$ ($Rcur(L)$) = numărul de cărţi nerge (roşii) care rămân în pachet după ghicirea primelor L cărţi. $Ncur(0)=N$ şi $Rcur(0)=R$. Pentru $L\geq 1$, dacă $Rcur(L-1)\geq Ncur(L-1)$, atunci $Rcur(L)=Rcur(L-1)-1$ şi $Ncur(L)=Ncur(L-1)$; altfel, $Rcur(L)=Rcur(L-1)$ şi $Ncur(L)=Ncur(L-1)-1$.

Dacă $L=N+R$, atunci $Nconfig(N+R)=1$. Altfel, jucătorul trebuie să nu ghicească a $(L+1)$-a carte. Dacă $Rcur(L)\geq Ncur(L)$, a $(L+1)$-a carte trebuie să fie neagră, deci setăm $Rcur2(L)=Rcur(L)$ şi $Ncur2(L)=Ncur(L)-1$; altfel, setăm $Rcur2(L)=Rcur(L)-1$ şi $Ncur2(L)=Ncur(L)$. $Nconfig(L)$ este egal cu $Ntotal(Ncur2(L)+Rcur2(L))$. Algoritmul este liniar, dar presupune folosirea operaţiilor pe numere mari, ceea ce nu este convenabil.

O soluţie mai simplă este următoarea. Vom calcula, pe rând, valorile $Prob(L)$, începând de la 0 şi până la $N+R$. Vom menţine, pe parcurs, valorile $Pcur$, $Ncur$, $Rcur$ şi $Lcur$. Iniţial, $Pcur=1.0$, $Ncur=N$, $Rcur=R$ şi $Lcur=0$. Dacă $Rcur\geq Ncur$, jucătorul va alege, la momentul curent, culoarea roşie; altfel, alege culoarea neagră. Probabilitatea de a ghici la momentul curent este $P=max\{Rcur,Ncur\}/(Rcur+Ncur)$. Dacă nu ghiceşte, atunci jocul se termină şi el rămâne cu $Lcur$ cărţi ghicite. $Pcur$ este probabilitatea de a ghici primele $Lcur$ cărţi. $Prob(Lcur)$ va fi egal cu $Pcur\cdot(1-P)$ (probabilitatea de a ghici $Lcur$ cărţi şi de a nu o ghici pe a $(Lcur+1)$-a).

După considerarea cazului în care nu a ghicit cartea, vom trece la pasul următor. Dacă $Rcur\geq Ncur$, setăm $Rcur=Rcur-1$; altfel, $Ncur=Ncur-1$ (deoarece cartea ghicită este scoasă din pachet). Apoi setăm $Pcur=Pcur\cdot P$ şi $Lcur=Lcur+1$. Când ajungem în starea $Ncur=Rcur=0$, setăm $Prob(Lcur=N+R)=Pcur$. În felul acesta, am determinat probabilităţile $Prob(L)$ pentru fiecare număr de cărţi $0\leq L\leq N+R$. Putem să nici nu menţinem aceste probabilităţi separat, ci să calculăm suma ce reprezintă răspunsul pe măsură ce calculăm probabilităţile. Complexitatea algoritmului este $O(N+R)$.

Problema 10-3. Vaci şi Maşini (UVA)

Participi la o emisiune televizată şi în faţa ta sunt $X+Y$ ($1\leq X,Y\leq 10.000$) uşi. În spatele a X uşi se află câte o vacă, iar în spatele celorlalte Y se află câte o maşină. În prima etapă, tu alegi una din uşi. Apoi, prezentatorul emisiunii deschide Z ($0\leq Z\leq X-1$) uşi în spatele cărora se află (sigur) o vacă (uşa pe care ai ales-o tu nu se află în mod cert printre cele Z deschise). După deschiderea celor Z uşi, prezentatorul îţi oferă posibilitatea să îţi schimbi alegerea (adică să alegi altă uşă). Tu accepţi oferta şi alegi o uşă diferită de cea aleasă iniţial (şi de cele Z deschise de prezentator). Care este probabilitatea ca în spatele uşii alese la final să găseşti o maşină ? (în mod clar, o maşină este ceea ce îţi doreşti, şi nu o vacă).
Exemplu: X=2, Y=1, Z=1 => Răspunsul este 0.666666.

Soluţie: Răspunsul este suma a doua probabilităţi, P_1 şi P_2. P_1 corespunde cazului în care în spatele uşii alese iniţial se afla o maşină, iar P_2 corespunde cazului în care în spatele uşii alese iniţial se află o vacă. $P_1=(Y/(X+Y))\cdot((Y-1)/(X-Z+Y-1))$. $P_2=(X/(X+Y))\cdot(Y/(X-Z-1+Y))$.

Problema 10-4. Zi de Naştere (UVA)

Stai la o coadă la cinematograf pentru a cumpara bilete la cel mai nou film cu actorul (actriţa) preferat(ă). De fiecare dată când cineva cumpără un bilet, vânzătorul îşi notează ziua de naştere a persoanei şi o compară cu zilele de naştere ale persoanelor care au cumpărat bilet înaintea ei (la care se adaugă ziua de naştere a vânzătorului, pentru a oferi o şansă şi

primei persoane aflate la coadă). Dacă ziua de naştere coincide cu a unei persoane care a cumpărat deja bilet (sau cu a vânzătorului), persoana care tomai a cumpărat biletul câştigă o invitaţie la cină din partea actorului preferat (actriţei preferate). Pe ce poziţie ţi-ai dori să te afli în coadă, pentru ca probabilitatea ca tu să câştigi invitaţia să fie maximă ? Determinaţi atât poziţia cu virgulă, cât şi poziţia întreagă. Poziţiile se numerotează de la *1*, iar un an are *Z* ($1 \le Z \le 100.000$) zile.
Exemplu: Z=365 => Răspunsul este 18.61 (iar poziţia cu număr întreg este 19).

Soluţie: Poziţia optimă (ca număr real) este $poz=(1+(4 \cdot Z+1)^{1/2})/2-1$. Poziţia ca număr întreg se obţine prin rotunjirea valorii reale (în jos, dacă partea fracţionară este mai mică decât *0.5*; sau în sus, pentru ≥ 0.5). Încercaţi să demonstraţi formula. Observaţi că probabilitatea *Prob(p)* de a câştiga invitaţia dacă vă aflaţi pe poziţia *p* este egală cu: $(1-Prob(1)-...-Prob(p-1)) \cdot p/N$ (cu $Prob(1)=1/N$). O altă expresie a lui *Prob(p)* este $p/N \cdot C(N,p)/p^N = C(N,p)/(N \cdot p^{N-1})$. (unde *C(a,b)* reprezintă combinări de *a* luate câte *b*).

Problema 10-5. Extragerea bilelor colorate

Într-o urnă se află bile din *N* culori ($1 \le N \le 10$); câte *b(i)* bile din culoarea *i* ($0 \le b(i) \le 10$). Dumneavoastră extrageţi *K* bile din urnă ($0 \le K \le b(1)+\ldots+b(N)$). Care este probabilitatea să fi extras *a(1)* bile de culoarea *1, ..., a(i)* bile de culoarea *i*, ... ($1 \le i \le N$; $a(1)+\ldots+a(N)=K$; $0 \le a(i) \le b(i)$) ?

Soluţie: Vom calcula *P(c(1), ..., c(N))*=probabilitatea de a fi extras exact *c(i)* bile de culoarea *i* ($1 \le i \le N$) între primele $c(1)+...+c(N)$ bile extrase. Avem $P(0, ..., 0)=1$. Vom calcula aceste valori în ordine lexicografică a şirurilor *(c(1), ..., c(N))*. Pentru a calcula *P(c(1), ..., c(N))*, iniţializăm această valoare cu *0*. Calculăm apoi numărul de bile rămase în urnă după extragerea celor *c(i)* bile din fiecare culoare *i* ($1 \le i \le N$): $NB=(b(1)-c(1))+...+(b(N)-c(N))$. Apoi considerăm, pe rând, toate culorile *i* ($1 \le i \le N$) cu $c(i)>0$ şi adunăm la *P(c(1), ..., c(N))* valoarea $(b(i)-c(i)+1)/(NB+1) \cdot P(c(1), ..., c(i-1), c(i)-1, c(i+1), ..., c(N))$; valoarea adunată reprezintă probabilitatea de a ajunge în starea *(c(1), ..., c(N))*, dacă la ultima extragere am extras o bilă de culoarea *i*.

Capitolul 11. Probleme Ad-Hoc

Problema 11-1. Detecţia repetiţiilor în şiruri recurente

Să considerăm un şir infinit, în care al *N*-lea element al şirului, $x(N)$ este dat ca o funcţie a ultimilor $K \geq 1$ termeni: $x(N)=f(x(N-1), ..., x(N-K))$. Primii *K* termeni, $x(1), ..., x(K)$ sunt daţi. Se pot defini multe probleme pe baza acestor şiruri, care, în general, se bazează pe detecţia repetiţiilor. De exemplu, dacă valorile elementelor şirului sunt numere întregi şi se ştie că sunt într-un interval $[A,B]$, atunci putem menţine o matrice *K*-dimensională $T(v(1), v(2), ..., v(K))$, cu $A \leq v(i) \leq B$. Dimensiunea matricii *T* este $(B-A+1)^K$. Această matrice este iniţializată cu *0*. Setăm apoi $T(x(1), ..., x(K))=K$, apoi începem să generăm, în ordine, elementele $x(i)$ ($i \geq K+1$). După ce am generat un element $x(i)$, verificăm dacă $T(x(i-K+1), ..., x(i))$ este nenul. Dacă nu este, atunci am găsit un ciclu de lungime $i-T(x(i-K+1), ..., x(i))$ în generarea valorilor şirului. Dacă este zero, atunci setăm $T(x(i-K+1), ..., x(i))=i$. Să presupunem că am detectat un ciclu de lungime *L*, care începe de la o poziţie *P* încolo. Atunci putem genera foarte simplu al *N*-lea element, unde *N* este foarte mare. Dacă $N \leq P$, atunci îl vom genera normal (în timp $O(P)$). Altfel, $x(N)=x(P+((N-P) \bmod L))$ (pe care îl generăm în timp $O(P+L)$).

O altă metodă pentru detecţia ciclurilor, foloseşte doi pointeri, p_1 şi p_2. p_1 este iniţializat la primul element, iar p_2 la al doilea. La fiecare iteraţie, p_1 avansează cu un pas (trece la elementul următor) şi p_2 avansează doi paşi. Pentru fiecare pointer se menţin, în ordine, ultimele *K* valori din şir peste care a trecut (inclusiv cea curentă). Dacă, la un moment dat, cele *K* valori corespunzătoare lui p_1 şi p_2 sunt identice, atunci am identificat un ciclu. Pentru a determina o poziţie de început a ciclului, vom porni cu un pointer p_3 de la începutul şirului şi vom avansa pas cu pas, până când ultimele *K* valori generate sunt cele corespunzătoare lui p_1 şi p_2. Totuşi, este posibil ca poziţia pe care o indică p_3 să nu fie chiar prima poziţie din ciclu, el putând să înceapă mai devreme. Dacă am putea merge înapoi cu câte un pas, atât cu p_3, cât şi cu p_1, atunci, atâta vreme cât ultimele *K* valori corespunzătoare celor doi pointeri ar fi identice, am merge înapoi cu un pas cu fiecare pointer. În felul acesta, am ajunge chiar la prima poziţie din ciclu.

Dacă dorim să găsim poziţia exactă de început a ciclului şi nu putem merge cu pointer-ii înapoi, putem proceda după cum urmează. După ce am găsit poziţia p_3 unde ultimele *K* elemente sigur aparţin ciclului, putem căuta binar poziţia p_4 ($1 \leq p_4 \leq p_3$), pentru a găsi poziţia exactă de început a ciclului (prima poziţie *poz* pentru care ultimele *K-1* elemente, plus elementul de pe poziţia *poz*, fac parte din ciclu). Să presupunem că am ales o poziţie *poz*. Vom genera, pornind de la începutul şirului, toate elementele, până ajungem la poziţia *poz*. Aici vom păstra ultimele *K* elemente. Vom parcurge şirul cu un pointer p_5, începând de unde a ajuns pointer-ul p_3. Ştim sigur că pointer-ul p_5 parcurge elemente de pe ciclu. Dacă, în timpul parcurgerii cu p_5, întâlnim, în ordine, ultimele *K* elemente indicate de poziţia *poz*, atunci poziţia *poz* face parte din ciclu şi, în cadrul căutării binare, vom încerca să găsim (dacă există) o poziţie mai mică. Dacă, însă, parcurgând şirul cu pointer-ul p_5, ajungem înapoi la elementele corespunzătoare pointer-ului p_3, fără a întâlni cele *K* elemente indicate de poziţia *poz*, atunci poziţia *poz* nu face parte din ciclu şi va trebui să testăm (în cadrul căutării binare) o poziţie mai mare.

Problema 11-2. Program Introspectiv (TIMUS)

Se consideră limbajul de programare PIBAS, descris în continuare. Un program PIBAS constă dintr-unul sau mai mulţi operatori separaţi prin „**;**”. Există două tipuri de operatori: operatorul de *atribuire* de şiruri şi operatorul de *afişare*. Operatorul de atribuire „**=**” este folosit în felul următor: ***<Variabila String>=<Expresie String>***. O <*Variabila String*> este o literă latină mare (*A-Z*). O <*Expresie String*> este ori o <*Variabilă String*>, o <*Constantă String*>, o <*Funcţie String*> sau o concatenare de <*Expresii String*>. Concatenarea a două <*Expresii String*> are forma: ***<Expresie String 1>+<Expresie String 2>***. O <*Constantă String*> este o secvenţă de caractere tipăribile, incluse între două ghilimele (”) sau două apostroafe (’). Această secvenţă nu poate conţine **”** sau **’** în interiorul ei, dacă este inclusă între acestea. De exemplu, *’repede’*, *”Olala!”* şi *"I don't know the solution."* sunt <*Constante String*> corecte.

O <*Funcţie String*> este folosită în felul următor: ***$(<Variabila string>,<întreg fără semn>,<întreg fără semn>)***. Funcţia întoarce un subşir (de caractere consecutive) al <Variabilei String>, incepând de la poziţia p (al doilea parameteru) şi având lungime *l* (al treilea parameteru). Primul caracter dintr-un şir se află pe poziţia *1*.

Operatorul de afişare este folosit astfel: ***?<Expresie String>***.

Exemple:

Program PIBAS	Ce afişază
?"Hello, "+'World!'	Hello, World!
A='World, Hello!';?$(A,8,5);?", ";B=$(A,1,5)+'!';?B	Hello, World!

Scrieţi un program PIBAS *introspectiv*. Un program este introspectiv dacă îşi afişază propriul cod sursă.

Soluţie: Aproape în orice limbaj de programare se pot scrie programe introspective. Limbajul PIBAS din cadrul problemei este unul simplu şi, în plus, faţă de un limbaj „standard”, are avantajul ca este un limbaj nou pentru toţi cei care încearcă să rezolve problema. Nu există o reţetă pe care să o puteţi folosi pentru a scrie un program introspectiv, de aceea, în continuare, este prezentat programul găsit de autor:

```
A=”’”;B=’”’;C=’=ABC;?$(C,2,1);?$(C,1,1);?B;?A;?B;?$(C,5,1);?$(
C,3,1);?$(C,1,1);?A;?B;?A;?$(C,5,1);?$(C,4,1);?$(C,1,1);?A;?C;
?A;?$(C,5,1);?$(C,6,128)’;?$(C,2,1);?$(C,1,1);?B;?A;?B;?$(C,5,
1);?$(C,3,1);?$(C,1,1);?A;?B;?A;?$(C,5,1);?$(C,4,1);?$(C,1,1);
?A;?C;?A;?$(C,5,1);?$(C,6,128)
```

Un program introspectiv în C (găsit la [Quine]) este următorul (se presupune ca header-ul *stdio.h* este inclus în mod implicit):

char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}.

Problema 11-3. Caraibe (Lot 2004, România)

N ($1 \le N \le 65.000$) piraţi au „obţinut” o sumă *S* foarte mare ($S > N \cdot (N-1)/2$). Pentru împărţire, piraţii se aşează în linie. Primul pirat va propune o schemă de împărţire a banilor. Dacă un anumit număr de piraţi nu sunt de acord cu această schemă, piratul va fi aruncat peste bord, şi apoi următorul pirat va propune o schemă de împărţire, şi tot aşa. Piraţii sunt foarte inteligenţi: un pirat este de acord cu o schemă de împărţire doar dacă aceasta îi aduce un avantaj strict (cel puţin un bănuţ) faţă de ce ar obţine votând împotriva schemei. Pentru că acţionează numai pe baze raţionale, piraţii sunt şi foarte predictibili. Cu alte cuvinte, un pirat poate anticipa decizia altor piraţi pentru a lua o decizie proprie (aceasta înseamnă şi că dacă

un pirat are mai multe posibilităţi de a alege o schemă de împărţire, ceilalţi piraţi ştiu ce variantă ar alege).

Depinzând de caracteristicile fiecărui pirat, numărul de piraţi care trebuie să fie de acord cu schema lui pentru a nu fi aruncat peste bord variază: pentru piratul i ($1 \le i < N$) acest număr este $A[i]$. Dacă piratul i propune o schemă, ştim că toţi piratii până la $i-1$ au fost aruncaţi deja peste bord. În afară de piratul i, mai există $N-i$ piraţi. Dacă cel puţin $A[i]$ dintre aceştia sunt de acord cu schema piratului i, comoara va fi împărţită după această schemă. Altfel, piratul i va fi aruncat peste bord, şi piratul $i+1$ va propune o schemă. Pentru orice i, avem $0 \le A[i] < N-i$. Datorită acestei condiţii $A[N-1]=0$, iar $A[N]$ nu este definit (pentru că piratul N este ultimul).

Primul pirat din linie doreşte să propună o schemă de împărţire a banilor astfel încât să nu fie aruncat peste bord, şi el să primească cât mai mulţi bănuţi. Determinaţi suma maximă pe care o poate primi. Se garantează că există o schemă pe care o poate propune primul pirat, astfel încât el să nu fie aruncat peste bord.

Soluţie: Ideea este să raţionăm pornind de la ultimul pirat spre primul. Dacă $N-2$ piraţi sunt aruncaţi peste bord, piratul $N-1$ îşi va acorda întreaga comoară lui. Piratul N nu poate să îl împiedice, pentru că $A[N-1]=0$. Deci piratul $N-3$ ştie ce se va întâmpla dacă el va fi aruncat peste bord ($N-1$ primeşte tot, şi N primeşte zero). Bazat pe asta, el poate propune o schemă care îi maximizează profitul şi este acceptată. Deci acum piratul $N-4$ ştie precis ce se va întâmpla dacă el este aruncat peste bord. Similar, prin inducţie, piratul $N-i-1$ ştie ce se va întâmpla dacă el este aruncat peste bord (va fi adoptată schema piratului $N-i$), şi poate lua o decizie bazată pe această certitudine.

Când propune o schemă, un pirat J vrea să îşi maximizeze profitul, dar este necesar ca schema lui să fie acceptată (altfel va fi aruncat peste bord). Pentru asta, el trebuie ca la $A[J]$ piraţi să dea mai mulţi bănuţi decât aceştia ar primii în schema piratului $J+1$. Pentru a-şi maximiza profitul, el alege să convingă în acest mod pe piraţii care primesc cele mai mici sume în schema lui $J+1$. La aceste sume, el adaugă câte un bănuţ. La ceilalţi piraţi, el le dă zero pentru că are deja suficienţi susţinători. Algoritmul va fi în esenţă acesta: considerăm toţi J de la $N-2$ până la 1; la pasul J ştim deja schema pe care ar propune-o $J+1$; pentru a afla schema piratului J, alegem $A[J]$ piraţi cu cele mai mici sume în schema precedentă, incrementăm câştigul acestor piraţi, şi facem câştigul celorlalţi egal cu zero.

Se observă că măcar suma unui pirat va fi facută zero (pentru că $A[J] < N-J$). Deci suma pe care o câştigă J este cel puţin suma maximă din suma lui $J+1$ minus $A[J]$ (fiindcă trebuie incrementate $A[J]$ sume). Pentru că numărul de bănuţi este foarte mare, reiese uşor că suma maximă din schema lui J este ce îşi acordă J lui însuşi. Mai precis, în schema proprie, J câştigă cel puţin $S–A[N-2] –A[N-1] – ... –A[J+1]$. Asta înseamnă că orice J poate propune o schemă în care să nu fie aruncat peste bord, şi mai mult poate câştiga mai mult decât orice pirat rămas.

Implementarea algoritmului este destul de uşoară şi are un timp de rulare $O(N)$. Ţinem o coadă cu piraţii în ordinea crescătoare a sumelor câştigate. Dacă mai mulţi piraţi câştigă aceaşi sumă, ţinem în coadă un singur element (numărul de piraţi care câştigă acea sumă). La pasul J, extragem din vârful cozii toţi piraţii care vor primii zero, până rămân doar $A[J]$ piraţi. Toţi aceşti piraţi vor fi introduşi la baza cozii ca un singur element (toţi primesc zero). Nu e nevoie să incrementăm explicit sumele pentru ceilalţi, ci putem să observăm că toţi din grupul I (unde primul grup din coadă are $I=0$) primesc I bănuţi, deci suma e dată de poziţia în coadă (care se schimbă în timp). Este uşor de văzut ca algoritmul rulează în timp $O(N)$: la

fiecare pas, în coadă se bagă doar un element (un grup de piraţi cu suma zero), deci numărul total de elemente scoase din coadă pe parcursul rulării este cel mult *N*.

Problema 11-4. Cutii (ACM ICPC NEERC Southern Subregion, 2003)

Se dau două cutii, conţinând *A*, respectiv *B* bile ($0 \le A,B \le 2.000.000.000$). O mutare constă în a transfera dintr-o cuitie în alta un număr de bile egal cu numărul de bile care se aflau în cutia destinaţie înaintea efectuării mutării. De exemplu, presupunând că $A \ge B$, după efectuarea mutării vom rămâne cu $A\text{-}B$, respectiv $2 \cdot B$ bile în cele două cutii. Determinaţi dacă este posibil să se transfere toate bilele dintr-o cutie în cealaltă şi, dacă da, după câte mutări.

Soluţie: Observăm că, la fiecare pas, există o singură mutare pe care o putem efectua şi anume, să transferăm bile din cutia cu mai multe bile în cea cu mai puţine bile. Aşadar, vom aplica acest algoritm şi vom număr mutările. Singura problemă ar fi să detectăm dacă nu se poate ajunge în configuraţia finală dorită. Să presupunem că, la un moment dat în cadrul algoritmului, am ajuns cu *A'* bile într-o cutie şi *B'* în cealaltă ($A' \ge B'$). Vom calcula *Q*, câtul împărţirii lui *A'* la *B'*. Dacă *Q* nu este un număr impar, atunci problema nu admite soluţie. Acest fapt se poate demonstra dacă pornim de la configuraţia finală (de fapt, cea aproape finală, în care se gaseşte un număr egal de bile în fiecare cutie) şi efectuăm mutările invers, considerând, de fiecare dată, cele două cazuri posibile. Dacă avem $A' \ge B'$ bile în cele două cutii, atunci înainte de efectuarea mutării puteam avea $A'+B'/2$ şi $B'/2$ bile (dacă şi *B'* este par), sau $A'/2$ şi $B'+A'/2$ (dacă şi *A'* este par). Observăm că în ambele cazuri, câtul împărţirii numărului mai mare la cel mai mic rămâne impar (am realizat, astfel, o demonstraţie prin inducţie după numărul de mutări efectuate în sens invers).

Problema 11-5. Expresie cu Împărţiri (Olimpiada Baltică de informatică, 2000)

Se dă următoarea expresie, ce conţine *N* ($2 \le N \le 100.000$) numere naturale: $a_1 / a_2 / ... / a_N$. Determinaţi dacă există vreo posibilitate de a insera *0* sau mai multe perechi de paranteze, astfel încât rezultatul expresiei să fie un număr natural.

Exemple:

N=4 **1/2/1/2**	**Răspuns: Da. O posibilitate de parantezare este următoarea: (1/2)/(1/2).**
N=3 **1/2/3**	**Răspuns: Nu.**

Soluţie: Observăm empiric că, în fracţia finală, ne-ar plăcea să nu aveam prea multe numere la numitor (şi cele mai multe să fie la numărătirul fracţiei). Observăm, de asemenea, că, indiferent de mmodul în care introducem paranteze, a_1 se va afla de fiecare dată la numărător, iar a_2 se va afla la numitor. Aşadar, o parantezare în care a_2 este singurul număr aflat la numitor ar fi cea mai convenabilă. Observăm că următoarea parantezare are această proprietate: $a_1/(a_2/a_3/.../a_N)$. Tot ce mai rămâne de făcut este să verificăm că produsul numerelor $a_1, a_3, a_4, ..., a_N$ este divizibil cu a_2. Bineînţeles, nu vom calcula produsl efectiv, deoarece ar putea avea foarte multe cifre. Vom parcurge numerele în ordinea $a_1, a_3, a_4, ...$; pentru fiecare număr a_i ($i=1$ sau $i=3,...,N$) vom calcula cel mai mare divizor comun d_i dintre a_i şi a_2, după care împărţim pe a_2 la d_i (modificăm, astfel, valoarea lui a_2). Dacă, la final, a_2 are valoarea *1*, atunci răspunsul este *„Da”*. În caz contrar, răspunsul este *„Nu”*.

Problema 11-6. Identificarea numărului lipsă

Se dau *N-1* numere distincte din mulţimea *{1,2,...,N}*. Determinaţi numărul lipsă (adică numărul din *{1,2,...,N}* care nu apare printre cele *N-1* numere date.

Soluţie: Dacă am avea suficientă memorie la dispoziţie, am putea scrie un program care să sorteze cele *N-1* numere. Am parcurge apoi numerele în ordinea sortată şi am verifica că fiecare număr este egal cu numărul anterior, plus *1*. Dacă nu este aşa, atunci numărul lipsă este chiar numărul anterior plus *1* (sau *N*, dacă am terminat de parcurs toate cele *N-1* numere). Această soluţie are complexitatea *O(N·log(N))*. O altă variantă de rezolvare ar consta în introducerea celor *N-1* numere într-o tabelă hash. Vom parcurge apoi toate numerele de la *1* la *N* şi, folosind tabela hash, vom verifica dacă un număr există sau nu printre cele *N-1* numere date. Această soluţie are complexitatea *O(N)*. Totuşi, dacă nu avem suficientă memorie la dispoziţie (dar chiar şi în cazul în care avem), următoarea abordare este cea mai eficientă. Calculăm suma *S* a celor *N-1* numere. Numărul lipsă va fi *N·(N+1)/2-S* (altfel spus, din suma *1+2+...+N* scădem suma *S*).

Problema 11-7. Identificarea numărului care apare de un număr impar de ori

Se dau *N* numere naturale *a(1), ..., a(N)*, având valori cuprinse între *0* şi *2.000.000.000*. Să considerăm mulţimea *M* a numerelor distincte dintre cele *N* date. Se ştie că fiecare număr din *M* apare de un număr par de ori printre cele *N* numere date, cu excepţia unuia singur, care apare de un număr impar de ori. Identificaţi numărul care apare de un număr impar de ori.

Soluţie: Evident, am putea sorta numerele şi apoi am parcurge şirul sortat. În felul acesta, am identfica uşor numărul care apare de un număr impar de ori. Această soluţie are complexitatea *O(N·log(N))*. O altă variantă ar consta în a menţine o tabelă hash, în care cheile sunt reprezentate de numere, iar valorile de numărul de apariţii al fiecăui număr. Parcurgem şirul şi dacă nu găsim un element *a(i)* în hash, introducem un element cu cheia *a(i)* şi valoarea *1*; dacă găsim un element *a(i)* cu valoarea *j*, ştergem acest element din hash şi introducem un element cu cheia *a(i)* şî valoarea *(j+1)*. Această soluţie are complexitatea *O(N)*. Ambele soluţii prezentate anterior folosesc memorie *O(N)*.

Putem rezolva problema şi folosind mai puţină memorie. Practic, vom calcula *x = a(1) xor a(2) xor ... xor a(N)*. Îl putem calcula pe *x* pe măsură ce citim numerele *a(1), ..., a(N)* (dintr-un fişier, de exemplu). *x* este numărul care apare de un număr impar de ori.

Problema 11-8. Identificarea numărului minoritar (Olimpiada Baltică de Informatică, 2004, enunţ modificat)

Se dau *N* numere naturale *a(1), ..., a(N)*, având valori cuprinse între *0* şi *2.000.000.000*. Să considerăm mulţimea *M* a numerelor distincte dintre cele *N* date. Se ştie că fiecare număr din *M* apare de exact $K \geq 2$ ori, cu excepţia unuia singur, care apare de $1 \leq Q \leq K-1$ ori. Vom numi acest număr minoritar. Identificaţi numărul minoritar, în condiţiile în care *K* şi *Q* nu sunt cunoscute.

Soluţie: Evident, soluţiile bazate pe sortare şi tabele hash pot fi folosite şi în acest caz, însă folosesc *O(N)* memorie. O soluţie cu memorie *O(1)* este următoarea. Vom parcurge fişierul de intrare şi vom găsi *2* numere distincte: *x* şi *y*. Dacă nu există *2* astfel de numere, atunci toate numerele *a(i)* sunt identice => *a(1)* este răspunsul. Apoi mai parcurgem fişierul o dată şi calculăm *nx* şi *ny*, numerele de apariţii în fişier ale lui *x* şi *y*; dacă *nx>ny*, răspunsul este *y*,

iar dacă $ny>nx$, răspunsul este x. Dacă $nx=ny$, atunci $K=nx$ şi am găsit valoarea lui K. Vom determina acum numărul minoritar bit cu bit (are cel mult *32* de biţi). Parcurgem fişierul de intrare încă o dată şi pentru fiecare bit b de la *0* la *31*, calculăm $num(b,q)$=numărul de numere din fişier pentru care bitul b are valoarea q ($q=0$ sau *1*). Pentru fiecare bit b, unul dintre numerele $num(b,0)$ şi $num(b,1)$ va fi multiplu de K, iar celălalt nu va fi. Dacă $num(b,q)$ este multiplu de K, atunci numărul minoritar are bitul b egal cu $(1-q)$. În felul acesta, putem determina fiecare bit al numărului căutat.

Problema 11-9. Dreptunghiuri Disjuncte (TIMUS)

Se dă o matrice de pixeli binari (*0* sau *1*) de dimeniuni NxN ($1\leq N\leq 5000$). Să se determine dacă componentele maximale (pe direcţiile Nord, Sud, Est sau Vest) de pixeli *1* (negri) sunt toate dreptunghiuri.

Soluţie: Este necesar ca orice sub-pătrat de dimensiuni *2x2* al matricii să conţină un număr X de pixeli de negri, unde $X\neq 3$ (X poate fi *0*, *1*, *2* sau *4*). Astfel, verificarea se poate realiza în timp $O(N^2)$ (timp proporţional cu dimensiunea matricii).

Problema 11-10. Depozit (TIMUS)

Într-un depozit există K ($1\leq K\leq 50.000$) seturi de produse. Un produs este identificat prin tipul acestuia, un număr de la *1* la N ($1\leq N\leq 100$). Spunem că două seturi S_1 şi S_2 sunt similare, dacă:

- S_1 este obţinut din S_2 prin eliminarea unui produs
- S_1 este obţinut din S_2 prin adăugarea unui produs
- S_1 (S_2) este obţinut din S_2 (S_1) prin înlocuirea unui produs cu un alt produs

De exemplu, setul *(1,2,3,4)* (conţine câte un produs din tipurile *1*, *2*, *3* şi *4*) este similar cu seturile *(1,2,3)*, *(1,2,3,4,5)*, *(1,2,2,3,4)*, *(1,3,4,5)*, etc. Am considerat produsele dintr-un set sortate în ordine crescătoare a tipurilor lor (deoarece ordinea acestor într-un set nu contează). Observaţi şi că un set poate conţine mai multe produse de acelaşi tip – practic, un set este definit prin numărul de produse din fiecare tip.

Împărţiţi seturile de produse date la cele M ($N<M$) magazine cu care depozitul are contract (fiecare set ajunge la un magazin), în aşa fel încât două seturi similare să nu ajungă la acelaşi magazin.

Soluţie: Vom calcula suma tipurilor produselor fiecărui set i, $S(i)$ ($1\leq i\leq K$). Observăm că dacă două seturi i şi j sunt similare, atunci $1\leq|S(i)-S(j)|\leq N$. De asemenea, observăm că un set nu este similar cu el însuşi, conform definiţiei (astfel, dacă există mai multe seturi care conţin acelaşi număr de produse din fiecare tip, ele pot fi distribuite la acelaşi magazin). Întrucât $M\geq N+1$, putem distribui fiecare set i la magazinul $((S(i) \bmod (N+1))+1)$. Complexitatea algoritmului este liniară în dimensiunea datelor de intrare.

Problema 11-11. Reconstrucţia unui şir din sumele a oricare 2 elemente

Se ştie că avem un şir ce constă din N valori ($3\leq N\leq 200$), pe care nu le cunoaştem. Cunoaştem, în schimb, cele $N\cdot(N-1)/2$ sume a oricare două elemente din şir. Determinaţi elementele şirului pe baza sumelor date.

Soluţie: Vom sorta şirul sumelor astfel încât să avem $S(1)\leq S(2)\leq..\leq S(N\cdot(N-1)/2)$. Să notăm elementele şirului prin $v(1), ..., v(N)$, astfel încât $v(1)\leq v(2)\leq..\leq v(N)$. În mod cert,

$S(1)=v(1)+v(2)$ şi $S(2)=v(1)+v(3)$. Mai trebuie să determinăm care dintre celelalte sume este suma $v(2)+v(3)$. Să observăm care ar putea fi valorile candidate: $S(3)$ ar putea fi $v(2)+v(3)$ sau $v(1)+v(4)$; dacă $S(3)$ nu este $v(2)+v(3)$, atunci $S(4)$ ar putea fi $v(2)+v(3)$ sau $v(1)+v(5)$; ... ; în general, $S(3 \le i \le N-1)$ ar putea fi egal cu $v(2)+v(3)$ sau $v(1)+v(i+1)$. Aşadar, avem $N-3$ sume candidate pentru a fi egale cu $v(2)+v(3)$. Vom considera fiecare din cele $O(N)$ posibilităţi. Să presupunem că am considerat că $v(2)+v(3)=X$. Vom construi un şir S' ce conţine elementele şirului S, din care eliminăm primele două elemente ($S(1)$ şi $S(2)$) şi pe X (dacă X apare de mai multe ori, se elimină o singură apariţie a sa). O dată ce am stabilit valoare X a lui $v(2)+v(3)$ avem un sistem de 3 ecuaţii cu 3 ncunoscute:

(1) $v(1)+v(2)=S(1)$;

(2) $v(1)+v(3)=S(2)$;

(3) $v(2)+v(3)=X$)

şi 3 necunoscute ($v(1), v(2), v(3)$). De aici putem obţine valorile lui $v(1)$, $v(2)$ şi $v(3)$. O dată ce valorile acestor elemente au fost determinate, ştim că primul element din şirul S' este $v(1)+v(4)$, obţinând, astfel, valoarea lui $v(4)$. După determinarea lui $v(4)$, vom elimina valorile $(v(1)+v(4))$, $(v(2)+v(4))$ şi $(v(3)+v(4))$ din şirul S'. Acum primul element (cel mai mic) din S' este egal cu $v(1)+v(5)$; ş.a.m.d. Atunci când ştim că cel mai mic element din S' este egal cu $v(1)+v(i)$ ($4 \le i \le N$), calculăm valoarea lui $v(i)$ (deoarece cunoaştem valoarea $v(1)$), apoi eliminăm din S' valorile $(v(j)+v(i))$ ($1 \le j \le i-1$); în continuare, dacă $i<N$, cel mai mic element din S' va fi egal cu $(v(1)+v(i+1))$. Dacă, la un moment dat, atunci când vrem să eliminăm o valoare Y din S' aceasta nu (mai) există în S', înseamnă că presupunerea iniţială (că $v(2)+v(3)=X$) este greşită şi va trebui să încercăm o altă valoare pentru $v(2)+v(3)$. Din punct de vedere al implementării, S' poate fi implementat ca un arbore echilibrat. În felul acesta complexitatea soluţiei este $O(N^3 \cdot log(N))$. Dacă diferenţa dintre suma minimă şi cea maximă nu este prea mare, atunci putem folosi un vector de frecvenţe ($f(y)$=numărul de apariţii în S' a valorii $S(1)+y$, $0 \le y \le S(N \cdot (N-1)/2)-S(1)$) şi complexitatea algoritmului se reduce la $O(N^3+VMAX)$, unde $VMAX$ este diferenţa dintre valoarea maximă şi minimă a unei sume dintre cele $N \cdot (N-1)/2$ date.

Problema 11-12. Şir circular

Se dă un şir circular ce constă din N elemente: $1, 2, ..., N$. Iniţial ne aflăm poziţionaţi pe elementul $e(1)$. De fiecare dată când ajungem pe o poziţie i, trebuie să înaintăm de $x(i) \ge 0$ ori în sens crescător (dacă $s(i)=1$) sau descrescător dacă ($s(i)=-1$). Următoarele $y(i)>0$ poziţii (începând de la cea pe care am ajuns după cele $x(i)$ înaintări) trebuie eliminată, după care trecem pe elementul următor (în sensul în care am efectuat parcurgere până atunci). Afişaţi elementele în ordinea în care acestea sunt eliminate.

Soluţie: O primă soluţie constă în menţinerea unor valori *next* şi *prev* pentru fiecare poziţie i. $next(i)$ este poziţia care urmează după i în sens crescător, iar $prev(i)$ este poziţia care urmează după i în sens descrescător. Vom menţine valoarea curentă *poz*, iniţial 1, precum şi numărul de elemente rămase M (iniţial $M=N$). Cât timp $M>0$ vom efectua următoarele acţiuni. Vom iniţializa două contoare $k=(x(poz) \bmod M)$ şi $l=y(poz)$, şi un sens $sk=s(poz)$. Apoi vom efectua atribuirea $poz=next(poz)$ ($poz=prev(poz)$) de k ori dacă $sk=1$ ($sk=-1$). După aceea, cât timp $l>0$, salvăm poziţia următoare poz' ($poz'=next(poz)$, dacă $sk=1$, sau $prev(poz)$, dacă $sk=-1$) şi ştergem poziţia *poz* din şir (o ştergere se realizează astfel: $next(prev(poz))=next(poz)$; $prev(next(poz))=prev(poz)$). După ştergere decrementăm M cu 1, decrementăm l cu 1, şi setăm $poz=poz'$. Dacă valorile $x(*)$ sunt mari, această soluţie are

complexitatea $O(N^2)$. Pentru acest caz, putem folosi un arbore de intervale pentru a găsi mai rapid următoarea poziţie ce trebuie ştearsă. De exemplu, să presupunem că înaintăm în sens crescător de *k* ori de la poziţia *poz*. Vom calcula (folosind arborele de intervale) câte poziţii *q* mai sunt active în intervalul *[poz+1,N]*. Dacă sunt mai puţin de *k*, setăm *k=k-q* şi mutăm *poz* la prima poziţie existentă mai mare sau egală cu *1* (o putem găsi în timp logaritmic, dacă menţinem în fiecare nod al arborelui de intervale câte poziţii active (neeliminate) există în subarborele asociat acestuia). Pentru a determina a *k*-a poziţie activă din intervalul *[poz,N]*, parcurgem arborele de intervale de la rădăcină spre frunze. Dacă fiul stânga al nodului la care am ajuns conţine *qleft≥k* poziţii active, mergem în fiul stânga; altfel mergem în fiul dreapta, dar nu înaintea de a seta *k=k-qleft*. Cazul parcurgerii în sens descrescător se tratează în mod similar. O ştergere a unei poziţii *poz* se realizează prin decrementarea cu *1* a numărului de poziţii active din toate nodurile de pe drumul de la frunza corespunzătoare intervalului de poziţii *[poz,poz]* până la rădăcina arborelui de intervale. Astfel, obţinem o complexitate *O(N·log(N))*. Putem folosi şi o împărţire în *O(sqrt(N))* grupuri de câte *O(sqrt(N))* poziţii, caz în care menţinem, pentru fiecare grup, câte poziţii active există în grupul respectiv. În felul acesta, putem găsi în timp *O(sqrt(N))* a *k*-a poziţie la care trebuie să ajungem (căutăm liniar în grupul ce conţine poziţia *poz*; apoi, atâta timp cât *k≥0*, sărim peste câte un grup întreg, decrementând *k* cu numărul de poziţii active din grup; dacă *k* devine *≤0* anulăm ultima modificare a lui *k* şi căutăm liniar poziţia dorită în cadrul grupului la care am ajuns).

Există şi alte variante ale problemei prezentate, însă, în afara unor cazuri particulare, cele două tehnici prezentate (cea bazată pe *next/prev* şi cea bazată pe arbori de intervale sau pe împărţirea în *O(sqrt(N))* grupuri de câte *O(sqrt(N))* poziţii, pot fi folosite pentru a rezolva orice astfel de problemă.

Problema 11-13. Lume liniară (ACM ICPC SEERC 2005)

Se dă o lume sub forma unui segment de dreaptă orizontal de lungime *L* (capătul stâng este la *x=0* şi capătul drept la *x=L*). În această lume se află *N* persoane care se deplasează cu aceeaşi viteză *V*. La momentul *t=0*, fiecare persoană *i* se află la coordonata *x(i)* (*0≤x(i)≤L*) şi îşi începe deplasarea în sensul *s(i)* (*s(i)=+1*, pentru sensul crescător al coordonatelor *x* şi *-1* pentru sensul descrescător), *1≤i≤N*. Când *2* persoane se întâlnesc, una venind din sensul crescător iar cealaltă din sensul descrescător, acestea îşi schimbă instantaneu sensul. Când o persoană ajunge la marginea segmentului (din stânga sau din dreapta), aceasta cade de pe segment şi dispare. Determinaţi care persoană cade ultima şi după cât timp. Se garantează că nu există două persoane care să cadă de pe segment simultan şi că oricare două persoane au coordonatele *x(i)* distincte.

Soluţie: Observăm că atunci când *2* persoane se întâlnesc este ca şi cum acestea ar trece una pe lângă cealaltă, deoarece fiecare continuă traiectoria celeilalte. Pentru a determina după cât timp cade ultima persoană, vom calcula pentru fiecare persoană *i*, valoarea *T(i)*=momentul de timp la care această persoană ar cădea de pe segment, în condiţiile în care trece pe lângă persoanele pe care le întâlneşte (adică nu îşi schimbă sensul la fiecare întâlnire). Dacă *s(i)=1*, atunci *T(i)=(L-x(i))/V*; dacă *s(i)=-1*, atunci *T(i)=x(i)/V*. Durata de timp *Tmax* după care cade ultima persoană este *max{T(i)}*. În mod similar, durata de timp după care cade prima persoană este *min{T(i)}*.

Pentru a determina care este persoana care cade ultima, vom proceda în felul următor. Vom sorta coordonatele *x(i)*, astfel încât să avem *x(o(1))<x(o(2))<...<x(o(N))*. Să definim

$po(i)$=poziţia pe care apare persoana i în această ordonare ($o(po(i))=i$). Pentru $0 \le t \le Tmax$, putem determina poziţiile $xp(i,t)= x(i)+s(i)\cdot V\cdot t$. La momentul t, va exista câte o persoană la fiecare din poziţiile $xp(i,t)$ ($1 \le i \le N$) (presupunând că segmentul ar avea lungime inifinită şi nu ar cădea nimeni). Observăm că, întrucât atunci când doi oameni se întâlnesc aceştia îşi schimbă sensul, ordinea relativă a celor N persoane (considerând coordonatele lor x) rămâne mereu aceeaşi. Aşadar, dacă sortăm poziţiile $xp(i,t)$ astfel încât să avem $xp(q(1),t) \le xp(q(2),t) \le .... \le xp(q(N),t)$, atunci persoana $o(i)$ se va afla la momentul t la coordonata $xp(q(i),t)$ ($1 \le i \le N$). Să considerăm persoana px pentru care $Tmax=T(px)$. Fie $pq(px)$ poziţia lui $xp(px, Tmax)$ în ordonarea $xp(q(1),Tmax) \le ... \le xp(q(N),Tmax)$, adică $q(pq(px))=px$. La momentul $Tmax$, la coordonata $xp(px,Tmax)=0$ sau L (unul din capetele segmentului) se va afla persoana $o(pq(px))$. Se observă uşor că algoritmul descris are complexitatea $O(N\cdot log(N))$.

Problema 11-14. Regele Alb-Negru (SGU)

Pe o tablă de şah de dimensiuni NxN ($2 \le N \le 10^6$) se află amplasaţi (în poziţii distincte) *3* regi: regele alb, regele negru şi regele alb-negru. Regele alb-negru este invizibil şi reprezintă o ameninţare pentru regele alb şi regele negru. Regele alb şi regele negru vor să se întâlnească pentru a stabili o alianţă împotriva regelui alb-negru. Pentru aceasta, cei *2* regi (alb şi negru) trebuie să ajungă în două poziţii alăturate (pe orizontală, verticală sau diagonală). Mutările încep la momentul de timp *0* şi alternează în felul următor. Întâi mută regele alb, apoi regele negru, apoi regele alb-negru, iar apoi ciclul de mutări se reia (fiecare rege se poate muta într-o poziţie alăturată pe orizontală, verticală sau diagonală, cu condiţia să nu părăsească tabla de şah). Regele alb şi regele negru se deplasează unul spre altul pe unul din drumurile de lungime minimă de pe tablă. Regele alb-negru se deplasează către unul din cei doi regi. Dacă în urma unei mutări efectuate de regele alb-negru, acesta ajunge pe o poziţia ocupată de regele alb sau de cel negru, atunci regele respectiv este omorât (şi alianţa este, astfel, împiedicată). Dacă regele alb sau regele negru, în urma mutărilor lor, ajung pe poziţia ocupată de regele alb-negru, atunci nu se întâmplă nimic. Regii nu au voie să stea pe loc atunci când le vine rândul să mute. În plus, regele alb-negru se deplasează conform următoarei proprietăţi: dacă, la un moment dat, se află în poziţia (L,C), atunci el a urmat unul din drumurile de lungime minimă din poziţia sa iniţială până la poziţia (L,C).

Considerând că regele alb-negru i-ar putea influenţa pe regele alb şi pe cel negru să aleagă unul din drumurile de lungime minimă alese de regele alb-negru, determinaţi numărul minim de mutări efectuate de regele alb-negru după care unul din cei doi regi (alb sau negru) moare (dacă acest lucru este posibil).

Soluţie: Fie (LA,CA), (LB,CB) şi (LAB,CAB) coordonatele *(linie,coloană)* ale regelui alb, regelui negru şi regelui alb-negru. Vom roti şi/sau oglindi tabla de şah şi/sau vom interschimba liniile cu coloanele, după cum este necesar, pentru a ajunge în situaţia: $CA<CB$, $LA \le LB$ şi $CB-CA \ge LB-LA$. Astfel, la fiecare mutare a regelui alb, coloana pe care se află acesta va creşte, iar la fiecare mutare a regelui negru, coloana pe care se află acesta va scădea. Numărul de poziţii intermediare dintre poziţia regelui alb şi cea a regelui negru este $D=CB-CA-1$. Numărul maxim de mutări pe care le poate efectua regele alb-negru este $Dmax=(D-1)$ *div 2*.

Vom încerca, pe rând, fiecare număr de mutări x ($1 \le x \le Dmax$), în ordine crescătoare, şi vom verifica dacă este posibil ca regele alb-negru să omoare pe regele alb sau pe regele negru exact la a x-a mutare a regelui alb-negru. Pe drumurile pe care le urmează, regele alb şi

regele negru se află mereu în interiorul unui paralelogram având două laturi paralele cu liniile tablei de şah şi două laturi paralele cu diagonalele secundare. Fie *Dif=(CB-CA)-(LB-LA)*. Latura de sus (de pe linia cu indice mai mic) conţine segmentul orizontal dintre poziţiile *(LA, CA)* şi *(LA, CA+Dif)*. Latura de jos (de pe linia cu indice mai mare) conţine segmentul orizontal dintre poziţiile *(LB, CB-Dif)* şi *(LB, CB)*. Paralelogramul conţine toate poziţiile de pe conturul şi din interiorul său. Acest paralelogram poate fi şi degenerat: un segment orizontal (dacă *LA=LB*) sau un segment de pe o diagonală secundară, dacă *Dif=0*.

După *x* mutări efectuate, regele alb se află pe coloana *CA+x*, undeva în interiorul paralelogramului, iar regele *B* pe coloana *CB-x*, tot undeva în interiorul paralelogramului. Vom calcula segmentele verticale *VA* şi *VB* ce corespund intersecţiei unei coloane *C* (*CA+x* sau *CB-x*) cu interiorul (şi conturul) paralelogramului (aceste segmente reprezintă intervale de linii de pe coloanele respective). După *x* mutări, regele alb-negru se află pe conturul unui pătrat de latură $2 \cdot x+1$, având centrul în poziţia sa iniţială. Trebuie doar să verificăm dacă vreunul din segmentele *VA* şi *VB* (calculate anterior) intersectează conturul pătratului (îl ating cu unul din capete, îl intersectează efectiv sau se suprapun parţial sau total pe una din laturile verticale ale sale). Dacă intersecţia dintre *VA* sau *VB* cu conturul pătratului regelui alb-negru este nevidă, atunci există o posibilitate ca la a *x*-a mutare, regele alb-negru să omoare unul din ceilalţi doi regi. Dacă intersecţia este vidă, atunci vom trece la următoarea valoare a lui *x* (*x+1*).

Dacă nu am găsit o intersecţie nevidă pentru nicio valoare a lui *x*, atunci regele alb-negru nu poate împiedica întâlnirea regilor alb şi negru.

Problema 11-15. Băutura otrăvită (SGU)

Pe masa unui rege se află *N* ($1 \leq N \leq 100.000$) pahare cu vin, dintre care se ştie că unul a fost otrăvit (cu o otravă letală). Regele doreşte să identifice paharul cu vin otrăvit. Pentru aceasta, el are la dispoziţie un număr foarte mare de servitori. El poate alege o parte dintre servitori (fie *M* numărul lor) şi poate asigna fiecăruia o submulţime de pahare de vin. Apoi, fiecare servitor gustă din fiecare pahar din submulţimea asignată. Otrava nu omoară imediat, ci după o perioadă suficient de lungă de timp, astfel că orice servitor, chiar dacă bea la un moment dat din paharul cu vin otrăvit, va trăi suficient de mult pentru a bea din toate paharele din submulţimea asignată. Determinaţi numărul *M* minim de servitori, precum şi submulţimile asignate acestora, astfel încât regele să poată identifica paharul cu vin otrăvit. Identificarea se face după un timp suficient de lung (mai lung decât timpul în care şi-ar face efectul otrava), după care se va şti, pentru fiecare servitor ales *i* ($1 \leq i \leq M$), dacă a murit sau dacă a supravieţuit.

Soluţie: În mod clar, o soluţie posibilă este ca regele să aleagă *M=N* servitori şi fiecare să guste din câte un pahar (servitorul *i* din paharul *i*). În felul acesta, va muri un singur servitor, care va identifica în mod unic paharul cu otravă. Totuşi, problema poate fi rezolvată şi cu $M=ceil(log_2(N))$ servitori. Servitorului *i* ($1 \leq i \leq M$) *i* se va asigna submulţimea de pahare cu numere de ordine *j* care au proprietatea că *j* are al *i*-lea bit egal cu *1* (vom numerota biţii de la *1* la *M*). Astfel, dacă un servitor *i* moare, vom şti sigur că paharul otrăvit are al *i*-lea bit egal cu *1*; dacă servitorul *i* supravieţuieşte, atunci al *i*-lea bit al numărului de ordine al paharului otrăvit este *0*. În felul acesta, putem identifica fiecare bit al numărului de ordine al paharului otrăvit (şi, deci, putem calcula acest număr).

Problema 11-16. Numere idempotente (TIMUS)

Determinaţi toate numerele x ($0 \le x \le N$), astfel încât $x \cdot x = x$ ($mod\ N$), unde N ($6 \le N \le 1.000.000.000$) este produsul a două numere prime distincte.

Soluţie: Este clar că $x=0$ şi $x=1$ sunt soluţii pentru orice N. În afară de 0 şi 1, mai există exact alte 2 soluţii. Întâi vom determina cele 2 numere prime P şi Q, astfel încât $N=P \cdot Q$. Pentru aceasta, vom considera toate numerele P de la 2 la $sqrt(N)$. Dacă $N\ mod\ P=0$, atunci numărul Q este $N\ div\ P$. După ce am determinat numerele prime P şi Q, vom folosi algoritmul lui Euclid extins, pentru a determina coeficienţii a şi b, astfel încât $a \cdot P+b \cdot Q=1$. Unul din coeficienţi este pozitiv, iar celălalt negativ. Dacă $a>0$, atunci setăm $x_1=a \cdot P$ şi $x_2=(b+P) \cdot Q$. Dacă $b>0$, atunci setăm $x_1=b \cdot Q$ şi $x_2=(a+Q) \cdot P$. x_1 şi x_2 sunt celelalte 2 soluţii ale ecuaţiei date. Vom justifica acest lucru pentru $a>0$ (cazul $b>0$ este simetric, prin înlocuirea lui a cu b şi a lui P cu Q). Avem $a \cdot P=(1-b \cdot Q)$. $(a \cdot P)^2=a \cdot P \cdot (1-b \cdot Q)=a \cdot P-a \cdot b \cdot P \cdot Q=a \cdot P$ ($mod\ (P \cdot Q)$). Aşadar, ecuaţia este verificată. Egalitatea $a \cdot P+b \cdot Q=1$ poate fi scrisă sub forma $a \cdot P-P \cdot Q +b \cdot Q+P \cdot Q=1 \Rightarrow (a-Q) \cdot P+(b+P) \cdot Q=1$. Întrucât $|b|<P$, avem că $b+P>0$. Acum putem scrie $a'=b+P$ şi $b'=a-Q$, obţinând $a' \cdot Q+b' \cdot P=1$. Putem folosi aceeaşi justificare ca mai sus (pentru $a \cdot P$), pentru $a' \cdot Q$.

Capitolul 12. Probleme Interactive

Problema 12-1. Master Mind (Olimpiada Balcanică de Informatică 2001, enunţ modificat)

Se consideră un şir secret *S* de *N* ($1 \le N \le 5$) cifre, fiecare cifră luând valori de la *0* la *M-1* ($0 \le M \le 10$). Pentru a ghici şirul secret, puteţi pune întrebări în care să daţi ca parametru un şir *SA* de *N* cifre ales de dumneavoastră. Modulul cu care interacţionaţi vă răspunde cu următoarele informaţii:

- cazul *1*: *NC*=în câte poziţii coincid şirul secret *S* şi şirul *SA*
- cazul *2*: *NC*=în câte poziţii coincid şirul secret *S* şi şirul *SA*; în plus, fie *NCW*=numărul maxim de poziţii din *SA* care pot coincide cu poziţiile din *S*, considerând cea mai favorabilă permutare a cifrelor lui *SA* => se răspunde şi cu valoarea *NCE=NCW-NC*.

Determinaţi şirul secret folosind un număr cât mai mic de întrebări.

Soluţie: Să considerăm mulţimea *SC* a tuturor celor M^N şiruri candidate pentru a fi şirul secret. Cât timp $|SC|>1$ vom alege un şir candidat *SA*. Pe baza răspunsului primit, vom parcurge toate şirurile din *S* şi le vom elimina pe acele şiruri *S'* pentru care răspunsul nu s-ar potrivi cu şirul *SA* (în cazul *1*: *S'* este eliminat dacă nu are *NC* poziţii comune cu *SA*; în cazul *2*, *S'* este eliminat dacă nu are *NC* poziţii comune cu *SA* sau dacă numărul maxim de poziţii în care coincide cu *SA*, considerând orice permutare posibilă a cifrelor lui *S'*, nu este *NC+NCE*).

Este de dorit să ajungem cât mai repede ca mulţimea *SC* să conţină un singur şir (şirul secret). Pentru aceasta, este importantă modalitatea în care alegem la fiecare pas şirul candidat *SA*. Avem mai multe variante. Una din ele constă în a alege un şir oarecare din *SC*. A doua variantă constă în a considera fiecare şir *S''* din *SC* (sau chiar dintre toate şirurile posibile) şi a considera toate răspunsurile posibile dacă *S''* ar fi ales drept şirul candidat *SA* (aceste răspunsuri se obţin ca reuniune a răspunsurilor obţinute dacă am considera fiecare şir *S'''* din *SC* ca fiind şirul secret ; bineînţeles, se elimină răspunsurile duplicate din reuniune). Pentru fiecare răspuns calculăm care ar fi cardinalul lui *SC* după eliminarea şirurilor care nu ar corespunde cu acel răspuns; îi ataşăm lui *S''* o pondere egală cu cardinalul maxim al lui *SC* obţinut în fiecare din răspunsurile posibile. Vom alege drept şir candidat *SA* acel şir *S''* pentru care ponderea este minimă.

Dacă se cunosc informaţii suplimentare despre şirul secret *S* (de ex., conţine cel puţin *l(i)* şi cel mult *h(i)* cifre cu valoarea *i*, $0 \le i \le M-1$), atunci vom porni cu o mulţime *SC* redusă, ce va conţine doar acele şiruri care satisfac restricţiile suplimentare.

Problema 12-2. Platou (Lotul Naţional de Informatică, România, 2006)

Se consideră un şir de 2^N elemente care sunt numere întregi. Se definesc noţiunile: *(1) platou*=o secvenţă de elemente egale aflate în vector în poziţii consecutive; *(2) lungimea unui platou* ca fiind numărul de elemente care alcătuiesc un platoul.

Se ştie că în şirul considerat, lungimea maximă a unui platou este 2^K ($0 \le K \le N$) şi că există cel puţin un platou care are această lungime. Amplasarea unui platou este determinată de poziţia cea mai mică şi de poziţia cea mai mare dintre poziţiile elementelor care alcătuiesc platoul. Şirul nu este cunoscut. Tot ceea ce puteţi face este să puneţi întrebări de forma: care

este lungimea cea mai mare a unui platou din şir aflat între poziţiile p_1 şi p_2 ? ($1\le p_1\le p_2\le 2\cdot N$). Aveţi voie să puneţi maxim $N-K+3$ întrebări.

Soluţie: O soluţie foarte simplă este următoarea. Vom căuta binar cel mai mic indice *pmin*, astfel încât lungimea maximă a unui platou din intervalul de poziţii *[1,pmin]* este 2^K. *pmin* se poate afla în intervalul $[2^K,2^N]$. Dacă în cadrul căutării binare vrem să testăm o valoare *pcand*, întrebăm care este lungimea *L* a unui platou conţinut în intervalul *[1,pcand]*. Dacă $L\ge 2^K$, atunci $pcand\ge pmin$; altfel, $pcand<pmin$. După găsirea lui *pmin*, platoul se află între poziţiile $pmin-2^K+1$ şi *pmin*. Această soluţie pune $log_2(2^N-2^K+1)$ întrebări. O soluţie mai bună este următoarea, care constă din *2* etape:
1) Localizarea unui interval de poziţii $[p_1,p_2]$ de lungime cel mult $2\cdot 2^K=2^{K+1}$, în care se află un platou de lungime 2^K.
2) Localizarea platoului de lungime 2^K în interiorul intervalului.

Pentru etapa 1), putem folosi următorul algoritm, bazat pe căutare binară. Alegem un indice *pd*, care va avea semnificaţia că în stanga lui (inclusiv *pd*) se termină un platou de lungime 2^K şi un indice *ps* cu semnificaţia că în stanga lui (exclusiv *ps*) nu se termină nici un platou de lungime 2^K. Iniţial *ps=1*, $pd=2^N$. Într-o buclă *while* facem urmatoarea căutare :

cât timp ($pd-ps>2^K$)
    pmid=(ps+pd)/2;
    întrebăm de intervalul $[p_1=1,p_2=pmid]$ şi primim răspunsul *R*
    dacă $R=2^K$ atunci *pd=pmid* altfel *ps=pmid+1*

Numărul de paşi pentru această buclă e $log_2(2^N/2^K)=N-K$. Intervalul $[p_1=max\{pd-2\cdot 2^K+1, 1\}, p_2=pd]$ are proprietatea că sigur conţine un platou de lungime 2^K. Dacă $[p_1,p_2]=[1,2^K]$, atunci am găsit platoul pe primele 2^K poziţii. Altfel, mai avem de pus *3* întrebări suplimentare. Notăm cu $pmid=(p_1+p_2)/2$ şi efectuăm *2* interogări: $ls=întrebare(p_1, pmid)$ şi $ld=întrebare(pmid+1,p_2)$. Avem următoarele cazuri:

- dacă $ls=2^K$, atunci platoul se află între poziţiile p_1 şi *pmid*
- dacă $ld=2^K$, atunci platoul se află între poziţiile *pmid+1* şi p_2
- dacă $ls+ld=2^K$, atunci paltoul se află între poziţiile *pmid-ls+1* şi *pmid+ld*
- dacă $ls+ld>2^K$, atunci între poziţiile p_1 si p_2 se mai află (pe lângă platoul de lungime 2^K), o „bucată” dintr-un platou de lungime$\le 2^K$; platoul de lungime 2^K se află :
    - cazul *1*: între poziţia din stânga=*pmid-ls+1* şi poziţia din drepata=$pmid–ls+2^K$ (intervalul I_1); sau
    - cazul *2*: între pozitia din dreapta=*pmid+ld* şi poziţia din stânga=$pmid+ld-(2^K-1)$ (intervalul I_2)

Pentru a stabili care din aceste situaţii are loc, mai este necesară o întrebare (de ex., întrebăm pentru intervalul I_1 şi dacă obţinem ca răspuns chiar lungimea intervalului I_1, atunci suntem în cazul *1*; altfel, suntem în cazul *2*). Aşadar, pentru a rezolva etapa *2)* mai sunt necesare *3* întrebări. În total sunt necesare cel mult *N-K+3* întrebări. Pentru *N=20* şi *K=13*, această soluţie pune cel mult *10* întrebări, în timp ce prima soluţie descrisă pune *20* întrebări.

Problema 12-3. Maxim (Lotul Naţional de Informatică, România, 2006)

Se consideră un şir *x* cu *N* ($3\le N\le 1.000.000$) componente numere întregi. Se ştie că oricare două componente din şir sunt diferite. Elementul din poziţia *p* ($2\le p\le N-1$) este maxim local dacă este strict mai mare decât elementele din poziţiile *p-1* şi *p+1*. Ultimul element din şir este maxim local dacă este strict mai mare decât elementele din poziţiile *1* şi *N-1* iar primul element din şir este maxim local dacă el este strict mai mare decât elementele din

pozițiile 2 și N. Dumneavoastră nu cunoașteți șirul. Determinarea poziției unui element care este maxim local se face adresând întrebări comisiei. O întrebare constă în a preciza trei poziții din șir, fie acestea p_1, p_2 și p_3. La o întrebare $Q(p_1,p_2,p_3)$ puteți primi unul din următoarele 4 răspunsuri:

- *1*, dacă între cele trei elemente din x avem relațiile $x[p_1]<x[p_2]$ și $x[p_2]<x[p_3]$
- *2*, dacă între cele trei elemente din x avem relațiile $x[p_1]<x[p_2]$ și $x[p_3]<x[p_2]$
- *3*, dacă între cele trei elemente din x avem relațiile $x[p_1]>x[p_2]$ și $x[p_2]>x[p_3]$
- *4*, dacă între cele trei elemente din x avem relațiile $x[p_1]>x[p_2]$ și $x[p_2]<x[p_3]$

Puteți folosi cel mult *25* de întrebări.

Soluție: Să presupunem că am găsit trei poziții p_1, p_2 și p_3, reprezentând intervalul de poziții $[p_1,p_3]$ (cu p_2 între p_1 și p_3 în sensul de pe interval), astfel încât $x[p_1]<x[p_2]$ și $x[p_2]>x[p_3]$. Vom efectua o interogare pentru pozițiile p_1', p_2 și p_2', calculate astfel: $p_1'=(p_1+p_2)$ *div* 2 ; $p_2'=(p_2+p_3)/2$ și în funcție de răspunsul primit avem următoarele cazuri:

- *2* : aplicăm același raționament pentru pozițiile p_1', p_2, p_2', deci pentru intervalul $[p_1',p_2']$
- *1* : aplicăm același raționament pentru intervalul p_2, p_2', p_3, deci pentru intervalul $[p_2,p_3]$
- *3* : aplicăm același raționament pentru intervalul p_1, p_1', p_2, deci pentru intervalul $[p_1,p_2]$
- *4* : aplicăm același raționament pentru oricare din intervalele $[p_1,p_2]$ (cu p_1' în interval) sau $[p_2,p_3]$ (cu p_2' în interval)

Se observă că pentru oricare din cele patru cazuri intervalul de poziții a fost redus la jumătate. Astfel, în $log_2(N)$ pași putem găsi *3* poziții consecutive p_1, p_2 și p_3, unde p_2 este maxim local. Pentru a găsi un prim interval de poziții care respectă condiția $x[p_1]<x[p_2]$ și $x[p_3]<x[p_2]$, facem o primă interogare pentru pozițiile *1*, $n/3$ și $2\cdot n/3$ și în funcție de răspuns, avem următoarele cazuri:

- *1* : pozițiile inițiale (p_1,p_2,p_3) sunt $n/3$, $2\cdot n/3$ și *1*
- *2* : pozițiile inițiale (p_1,p_2,p_3) sunt *1*, $n/3$, $2\cdot n/3$
- *3* : pozițiile inițiale (p_1,p_2,p_3) sunt $2\cdot n/3$, *1*, $n/3$
- *4* : este necesară încă o interogare care să decidă care din numerele de pe pozițiile *1* sau $2\cdot n/3$ este mai mare (de ex., $Q(n/3,2\cdot n/3,1)$ => dacă răspunsul este *1* sau *4*, vom avea $(p_1,p_2,p_3)=(2\cdot n/3, 1, n/3)$; altfel, $(p_1,p_2,p_3)=(n/3, 2\cdot n/3, 1)$)

Problema 12-4. Dreptunghi

Să considerăm o rețea de $N\times N$ ($1\leq N\leq 31.000$) celule, în care colțul din stânga-sus are coordonatele carteziene (1,1). În interiorul acestei rețele se află un dreptunghi (pe care îl vom nota în continuare cu *D1*) cu laturi paralele cu axele de coordonate, compus din celule ale rețelei. Determinați amplasarea dreptunghiului *D1*, punând întrebări de tipul următor: dați ca argument al întrebării un dreptunghi *D2* (prin liniile și coloanele celulelor ce reprezintă colțurile stânga-sus și dreapta-jos), iar răspunsul la întrebare este numărul de celule din intersecția dreptunghiului căutat *D1* cu dreptunghiul *D2* dat ca argument. Aveți voie să puneți maxim *31* de întrebări.

Soluție: Se începe cu o întrebare cu toată suprafața, aflând astfel numărul *np* de celule din *D1*. Căutăm apoi binar linia de sus a unuia dintre colțuri, întrebând cu dreptunghiuri ce conțin toate coloanele și număr de linii variabil (intervalul ce corespunde valorilor candidate pentru linia de sus se înjumătățește la fiecare pas). Inițial, acest interval este $[1,N]$. Când

întrebăm cu primele *x* linii ale dreptunghiului. Fie *nr* numarul de puncte pe care le primim ca răspuns:

- dacă *nr=0*, atunci linia de sus a lui *D1* este $\geq x$ şi păstrăm jumătatea de interval ce conţine valorile mai mari decât *x*;
- dacă *nr>0*, atunci linia de sus a dreptunghiului *D1* este $\leq x$;

Ne oprim când intervalul nostru conţine o singură linie (adiă e de forma *[lin,lin]*). Vom reţine numărul de puncte *ncol* de la ultima întrebare cu răspuns strict pozitiv (iar linia de sus este linia *lin*). Dacă toate răspunsurile au fost *0*, atunci linia de sus este linia *lin*, ea conţinând *ncol=np* puncte.

După aceste prime $log_2(N)$ întrebări, putem deduce şi numărul de linii *nlin=np/ncol*. Dacă *ncol>1*, vom pune întrebări cu un dreptunghi *D2* având o singură linie (linia de sus a lui *D1*), coloana din stânga egală cu *1* şi coloana din dreapta variabilă. Căutăm, astfel, binar coloana din stânga a lui *D1*. Căutarea este similară primei etape. Dacă răspunsul este *0* pentru o coloană dreapta *y*, va trebui să căutăm valori mai mari ale lui *y*; dacă răspunsul este *>0*, coloana căutată este $\leq y$. În acest moment, după $2\cdot log_2(N)+1$ întrebări, am găsit un colţ al dreptunghiului şi dimensiunile acestuia, identificând, astfel, complet dreptunghiul.

O altă soluţie este următoarea. După întrebarea iniţială din care aflăm numărul de puncte *np* din dreptunghi, vom pune întrebări cu dreptunghiuri având *N* coloane. Vom menţine un interval *[A,B]* (începem cu *A=1* şi *B=N*). Pentru un interval *[A,B]*, vom pune o întrebare cu liniile cuprinse între *A* şi *(A+B) div 2*. Dacă răspunsul *nr* este $\geq np$, atunci setăm *B=(A+B) div 2*; dacă *nr=0*, atunci *A=((A+B) div 2) + 1*. Ne oprim când *A=B* sau răspunsul *nr* este între *1* şi *np-1* (inclusiv). În acest moment, suntem siguri că linia *((A+B) div 2)* taie dreptunghiul. Fie ultimul răspuns primit *nq* (şi *A<B*) şi linia care taie dreptunghiul o notăm prin *C*. Vom căuta binar linia de sus a dreptunghiului, în intervalul *[A'=A, C'=C]*. Vom pune întrebări cu dreptunghiuri având *N* coloane şi linii cuprinse între *(A'+C') div 2* şi *C'*. Dacă răspunsul *nr* este *<nq*, atunci setăm *C'=((A'+C') div 2)-1*; dacă *nr=nq*, atunci setăm *A'=((A'+C') div 2)+1*. Vom reţine indicele ultimei linii *Lsus=(A'+C') div 2* (pentru intervalul *[A',C']* considerat la momentul respectiv) pentru care răspunsul a fost egal cu *nq*. Astfel, am identificat linia de sus a dreptunghiului în $log_2(N)$ întrebări în total. Putem identifica şi numărul de coloane *ncol=nq/(C-Lsus+1)* şi, apoi, numărul de linii *nlin=np/ncol*. Apoi vom căuta coloana cea mai din stânga, punând alte $log_2(N)$ întrebări cu dreptunghiuri ce conţin *N* linii (în mod similar celui în care căutăm coloana stânga în soluţia anterioară). Şi această soluţie pune tot $2\cdot log_2(N)+1$ întrebări.

Problema 12-5. Meandian (Olimpiada de Informatică a Europei Centrale, 2006)

Se consideră un şir ce conţine *N* ($4\leq N\leq 1000$) elemente distincte: *a(1), ..., a(N)*. Avem la dispoziţie următoarea operaţie: *Meandian(i1, i2, i3, i4)*. Această operaţie sortează crescător elementele *a(i1), a(i2), a(i3), a(i4)* (să presupunem că ordinea lor este *a(j1)<a(j2)<a(j3)<a(j4)*, unde *{j1, j2, j3, j4}={i1, i2, i3, i4}*) şi întoarce media celor *2* elemente din mijloc *((a(j2)+a(j3))/2)*. Cei *4* indici *i1, ..., i4*, trebuie să fie diferiţi unul de altul. Folosind această operaţie de cel mult *10.000* de ori, determinaţi toate valorile *a(1), ..., a(N)* ce pot fi determinate (mai exact, pentru fiecare indice *i*, determinaţi valoarea *a(i)*, dacă aceasta poate fi determinată).

Soluţie: Dacă am considera valorile *a(i)* sortate crescător, atunci fie *i1* şi *i2* indicii celor mai mici *2* valori (*a(i1)* şi *a(i2)*), şi *i3* şi *i4* indicii celor mai mari *2* valori. Chiar dacă am cunoaşte indicii *i1* şi *i2* (respectiv *i3* şi *i4*), nu am putea determina exact care sunt valorile

a(i1) şi *a(i2)* (respectiv *a(i3)* şi *a(i4)*). Astfel, vom putea determina exact doar *N-4* valori din şirul *a(1), ..., a(N)*.

Să considerăm că am ales *5* indici: *i1, i2, i3, i4, i5* (şi să presupunem că avem *a(j1)<a(j2)<a(j3)<a(j4)<a(j5)*, unde *{j1, ..., j5}={i1, ..., i5}*; ordinea *j1, .., j5* nu este cunoscută). Vom pune întrebările următoare:

- *b1=Meandian(i1, i2, i3, i4)*
- *b2=Meandian(i1, i2, i3, i5)*
- *b3=Meandian(i1, i2, i4, i5)*
- *b4=Meandian(i1, i3, i4, i5)*
- *b5=Meandian(i2, i3, i4, i5)*

Vom sorta valorile *b1, ..., b5*, astfel încât să avem $c1 \le c2 \le c3 \le c4 \le c5$ (unde *c1, ..., c5*, sunt o permutare a valorilor *b1, .., b5*; iar c_i corespunde valorii *bp(i)*). Referindu-ne la valorile *j1, ..., j5* menţionate anterior, sunt evidente următoarele lucruri:

- *c1=Meandian(j1, j2, j3, j4)=(a(j2)+a(j3))/2*
- *c2=Meandian(j1, j2, j3, j5)=(a(j2)+a(j3))/2=c1*
- *c3=Meandian(j1, j2, j4, j5)=(a(j2)+a(j4))/2*
- *c4=Meandian(j1, j3, j4, j5)=(a(j3)+a(j4))/2*
- *c5=Meandian(j2, j3, j4, j5)=(a(j3)+a(j4))/2=c4*

Vom considera acum următorul sistem cu *3* ecuaţii şi *3* necunoscute:

2·c1=a(j2)+a(j3)
2·c3=a(j2)+a(j4)
2·c5=(a(j3)+a(j4))

Rezultă *a(j2)=2·c1-a(j3)*; *a(j4)=2·c5-a(j3)*. Înlocuind în a *2*-a ecuaţie, obţinem *2·(c1+c5)-2·a(j3)=2·c3 => a(j3)=c1+c5-c3*. Pentru a determina efectiv şi indicele *j3* (din mulţimea *{i1, ..., i5}*), alegem rezultatul *bp(3)=c3*. Ştim că *bp(3)* este rezultatul acelei întrebari *Meandian* care a implicat toţi ceilalţi indici în afară de *j3*. Astfel, *j3* este acel indice care nu a fost folosit în întrebarea în urma căreia s-a obţinut rezultatul *bp(3)=c3*.

Astfel, după *5* întrebări, am reuşit să determinăm o valoare dintre oricare *5* indici *i1, ..., i5*. Vom folosi această metodă pentru a determina toate cele *N-4* valori ce pot fi determinate, punând *5·(N-4)* întrebări. Vom menţine o mulţime *S*, pe care o iniţializăm la *S={1,...,5}*. Punem cele *5* întrebări pentru cei *5* indici din *S* şi determinăm indicele *j3* (şi *a(j3)*). Apoi, pentru *i=6,...,N* efectuăm următoarele acţiuni:

(1) S=S\{j3}+{i} (*j3* este indicele determinat la pasul anterior);
(2) determinăm indicele *j3* (şi valoarea *a(j3)*) considerând cei *5* indici din *S*.

Deci, în mod repetat, excludem din *S* indicele pentru care am determinat valoarea anterior şi adăugăm un nou indice.

Problema 12-6. Găseşte punctul (Olimpiada Naţională de Informatică, Croaţia, 2005)

Se consideră un spaţiu *d*-dimensional în care se află un punct special la coordonate întregi necunoscute. Ştim doar că el se află în interiorul paralelipipedului $[0,XMAX=10^9]^d$. Se doreşte identificarea locaţiei punctului special în felul următor. Aveţi la dispoziţie un punct al dumneavoastră, care se află iniţial la coordonatele întregi *(x(1), ..., x(d))*. Puteţi efectua mutări conform căror punctul dumneavoastră se deplasează la orice alte coordonate întregi *(x'(1), x'(2), ..., x'(d))*. După fiecare mutare veţi afla dacă noua poziţie a punctului dumneavoastră este mai aproape de poziţia punctului special faţă de poziţia anterioară a punctului dumneavoastră, sau mai depărtată (în caz de distanţă egală se poate răspunde

oricum). Distanţa folosită este cea eculidiană. Găsiţi locaţia punctului special folosind cel mult $60 \cdot d$ mutări.

Soluţie: Vom determina fiecare coordonată a punctului special independent de celelalte. Să presupunem că vrem să determinăm coordonata j ($1 \le j \le d$). Vom efectua o căutare binară după coordonata j, după cum urmează. Vom menţine un interval $[a,b]$ în care se poate afla coordonata j a punctului special. Iniţial, $a=0$, $b=XMAX$. Cât timp $b>a$ vom efectua următoarele acţiuni. Modificăm coordonata j punctului nostru astfel încât $x(j)=a$, apoi mutăm punctul nostru astfel încât $x(j)=b$ (coordonatele $j' \neq j$ rămân nemodificate). Dacă la mutarea în care am setat $x(j)=b$, punctul nostru este mai departe de punctul special, atunci vom seta $b=inf((a+b)/2)$; altfel, dacă pentru $x(j)=b$ punctul nostru este mai aproape de punctul special, atunci setăm $a=sup((a+b)/2)$. Am notat prin $inf(q)$=parte întreagă inferioară din q şi prin $sup(q)$=parte întreagă superioară din q. Astfel, vom efectua $2 \cdot log(XMAX)=60$ mutări pentru fiecare coordonată j ($1 \le j \le d$).

Problema 12-7. Monede (Lotul Naţional de Informatică, România, 2003)

Aveţi la dispoziţie N ($2 \le N \le 32768$; N par) monede, fiecare având unul din următoarele 3 tipuri: monedă de aur (1), monedă de argint (2) sau monedă de bronz (3). Se ştie că unul din cele 3 tipuri este majoritar (există cel puţin $N/2+1$ monede din tipul respectiv). Se doreşte găsirea unei monede din tipul majoritar prin efectuarea de un număr minim de ori a următoarei operaţii: se formează $N/2$ perechi de monezi (fiecare monedă face parte dintr-o pereche) şi se compară cele 2 monezi din cadrul fiecărei perechi. Rezultatul unei comparaţii poate fi 1 (cele 2 monezi sunt de acelaşi tip) sau 0 (cele 2 monezi sunt de tipuri diferite). Rezultatul unei operaţii este şirul rezultatelor celor $N/2$ comparaţii efectuate.

Soluţie: În prima rundă comparăm monedele $2 \cdot k-1$ cu $2 \cdot k$ ($1 \le k \le N/2$). Fie S mulţimea monedelor $2 \cdot k-1$ cu proprietatea că $2 \cdot k-1$ şi $2 \cdot k$ sunt identice. Organizăm S ca o listă circulară dublu-înlănţuită (elementele sunt aşezate în listă într-o ordine arbitrară). La runda 2, comparăm pe x cu $prev(x)$ şi $next(x)$ – unde $prev$ şi $next$ se referă la relaţia în lista dublu-înlănţuită. Deoarece lista e circulară, fiecare element din S apare în exact două comparaţii. Cum facem asta într-o singură rundă? Păi, ştim că pentru orice x din S, $x+1$ este egal cu x. Deci putem folosi x într-o comparaţie şi $x+1$ în cealaltă. Folosind rezultatele de la comparaţii, grupăm elementele din listă în grupuri de elemente consecutive care sunt egale; primul element din fiecare grup va fi reprezentantul grupului.

Organizăm reprezentanţii grupurilor într-o listă circulară dublu-înlănţuită. La a treia rundă comparăm x cu $prev(prev(x))$ şi $next(next(x))$ – unde $prev$ şi $next$ se referă acum la lista reprezentanţilor. Folosind rezultatele acestor comparaţii, putem afla tipul fiecărui reprezentant. Deci ştim tipul monedelor din fiecare grup, deci putem determina tipul majoritar în mod trivial (prin numărare). Dar cum ştim tipul reprezentanţilor ? Primelor două elemente le atribuim arbitrar tipurile 1 şi 2. În continuare examinăm elementele pe rând. Dacă x este egal cu $prev(prev(x))$, ştim tipul lui x pentru că deja ştim tipul lui $prev(prev(x))$. Dacă x este diferit de $prev(prev(x))$, mai ştim şi că el este diferit de $prev(x)$ (din modul cum am construit grupurile şi reprezentanţii), aşa că tipul lui x este unic determinat ca fiind singurul tip rămas (elementul din mulţimea $\{1,2,3\}$ din care eliminăm tipul lui $prev(x)$ şi pe cel al lui $prev(prev(x))$).

Putem demonstra că problema nu se poate rezolva în mai puţin de 3 runde în cazul cel mai defavorabil, construind un adversar care ne obligă să efectuăm 3 runde. Orice comparaţii

ar efectua algoritmul în prima rundă, adversarul răspunde „egal” la toate. Astfel se creează perechi de elemente care sunt egale pe care le imaginăm „unite” într-un singur element. La a doua rundă, algoritmul poate face maxim *2* comparaţii cu fiecare element unit (pentru că un astfel de element înseamnă o pereche de elemente). Interpretate ca un graf, comparaţiile cerute formează o reuniune de cicluri şi lanţuri disjuncte. Dacă graful nu e conex (există cel puţin două cicluri/lanţuri), adversarul are o strategie destul de simplă. Pentru orice componentă care conţine mai puţin de jumătate din noduri, adversarul răspunde „egal” la toate comparaţiile. Dacă există o componentă cu mai mult de jumate din noduri (evident aceasta este unică), adversarul răspunde că jumate minus *1* dintre monede sunt de acelaşi tip, şi celelalte sunt de alt tip. Dacă algoritmul încercă să dea răspunsul la problemă, se poate construi uşor un aranjament al monedelor care reprezintă un contraexemplu şi este consistent cu răspunsurile date de adversar. Deci algoritmul trebuie să mai efectueze cel puţin o rundă. Dacă graful este conex, adversarul răspunde „diferit” la toate comparaţiile. Din nou, dacă algoritmul încearcă să dea răspunsul la problemă, se poate construi uşor un contraexemplu care arată că răspunsul nu e adevărat. Deci algoritmul mai trebuie să efectueze o rundă. În consecinţă, este nevoie de minim *3* runde în cel mai defavorabil caz.

Capitolul 13. Transformări, Evaluări, Ecuaţii şi Sisteme de Ecuaţii

Problema 13-1. Ecu (Lotul Naţional de Informatică, România, 2003)

Gigel are de rezolvat un sistem complicat de ecuaţii neliniare şi pentru aceasta intenţionează să folosească o metodă iterativă care, speră el, va converge către soluţie după un număr rezonabil de iteraţii. Mai întâi, el alege nişte valori iniţiale pentru cele **N** necunoscute ale sistemului. Aceste valori se notează cu $x_1^{(0)}\ x_2^{(0)} \dots x_N^{(0)}$. În continuare, după fiecare iteraţie, el va modifica valorile necunoscutelor, conform următoarelor relaţii:

- $x_k^{(i)} = p_k * x_k^{(i-1)} + (1 - p_{k+1}) * x_{k+1}^{(i-1)} + y_k$, pentru $1 \le k < N$
- $x_N^{(i)} = p_N * x_N^{(i-1)} + (1 - p_1) * x_1^{(i-1)} + y_N$

unde prin $x_k^{(i)}$ s-a notat valoarea necunoscutei k după i iteraţii. p_k reprezintă ponderea asociată necunoscutei k, iar y_k reprezintă corecţia aplicată necunoscutei k, după fiecare iteraţie.

Dându-se valorile iniţiale ale celor N ($2 \le N \le 30$) necunoscute, ponderile asociate şi corecţiile aplicate, să se determine valorile necunoscutelor după M ($0 \le M \le 10^9$) iteraţii.

Soluţie: Să notăm cu *x(i,j)* valoarea necunoscutei *i* după j iteraţii. *x(i,0)* sunt date.
x(i,1) = p(i)·x(i,0) + (1-p(i+1))·x(i+1,0) + y(i)
x(i,2) = p(i)·x(i,1) + (1-p(i+1))·x(i+1,1) + y(i) =
p(i)·(p(i)·x(i,0) + (1-p(i+1))·x(i+1,0) + y(i)) +
p(i+1)·(p(i+1)·x(i+1,0) + (1-p(i+2))·x(i+2,0) + y(i+1)).

x(i,j) se poate scrie ca o funcţie liniară de valorile iniţiale. Astfel,
x(i,j) = c(i,j,1)·x(1,0) + c(i,j,2)·x(2,0) + .. + c(i,j,N)·x(N,0) + c(i,j,0) ,
unde *c(i,j,k)* cu *k* de la *0* la *N* sunt nişte constante reale. Evident, dacă am determina *c(i,M,k)* cu *k* de la *0* la *N* am putea afla valorea lui *x(i,M)* (exact ceea ce ne interesează).

Observăm că nu ne interesează toate valorile *c(i,j,k)*, ci numai valorile de tipul $c(i,2^p,k)$ (adică numai dupa un număr de iteratii egal cu o putere a lui *2*). Să considerăm că am calculat aceste valori. Atunci putem determina *c(i,M,k)*, parcurgând reprezentarea binară a lui M. Să considerăm că primul bit de *1* al lui *M* se află pe poziţia *p*. Atunci, *c(i,M,k)*=Suma de la *j=0* la *N* din $c(i,2^p,j)\cdot c(j,M-2^p,k)$ (este similar cu înmulţirea de matrici; de fapt, problema se poate rezolva şi considerând relaţii între matrici şi vectori). Rămâne de determinat $c(i,M-2^p,k)$, care se determină ca şi pentru *M* ($M-2^p$ este *M* fără cel mai semnificativ bit de *1*). Valorile $c(i,2^p,k)$ sunt egale cu:

$c(i,2^p,k)$=Suma de la *j=0* la *N* din $c(i,2^{p-1},j)\cdot c(j,2^{p-1},k)$.

Valorile *c(i,1,k)* sunt usor de determinat, din relatiile date în enunţ, căci *x(i,1) = c(i,1,i)·x(i,0)+c(i,1,i+1)·x(i+1,0)+c(i,1,0)*. Pentru a nu lucra efectiv cu termeni constanţi, se poate considera că există o necunoscută în plus, cu numărul *0*, a cărei valoare este egală cu *1* şi nu se modifică în cursul unei iteraţii. In schimb, relaţiile de calcul devin:
x(i,j) = p(i)·x(i,j-1)+(1-p(i+1))·x(i+1,j-1)+y(i)·x(0,j-1), unde *x(0,k)=1* ($k \ge 0$).

Astfel, complexitatea algoritmului este $O(N^3 \cdot log(M))$ şi memoria folosită este $O(N^2 \cdot log(M))$.

Problema 13-2. Expresii min-max (Happy Coding 2006, infoarena)

Consideraţi o expresie care conţine numere naturale, paranteze, şi operatorii binari *m* şi *M*. *m* este operatorul de minim şi *M* este operatorul de maxim. Astfel, rezultatul operaţiei *A m B* este valoarea minimă dintre *A* şi *B*, iar rezultatul operaţiei *A M B* este valoarea maximă dintre *A* şi *B*. De exemplu, rezultatul *2m7* este *2*, iar rezultatul *9M8* este *9*. Cei doi operatori au aceeaşi prioritate, ceea ce înseamnă că dacă nu sunt paranteze, vor fi evaluaţi, în ordine, de la stânga la dreapta. De exemplu, rezultatul expresiei *1M22m13m789* este *13*.

Dându-se o expresie care conţine numere naturale, paranteze şi aceşti doi operatori, aflaţi rezultatul expresiei. Expresia va conţine cel mult *100.000* de caractere, iar numerele din cadrul ei sunt între *0* şi *1.000.000*.

Soluţie: Expresia dată poate fi evaluată folosind o metodă de complexitate liniară ($O(N)$, unde N este lungimea expresiei). Pentru simplificarea explicaţiilor, vom considera că întreaga expresie este încadrată într-o pereche de paranteze. Se parcurge expresia de la stânga la dreapta şi se va menţine o stivă cu operatorii neevaluaţi încă, cu rezultate parţiale şi cu paranteze deschise. Când se intâlneşte un operator, acesta se adaugă în stivă. Când se întâlneşte un operand (număr) şi în vârful stivei este un operator, se efectuează operaţia respectivă (câci în stivă se va găsi şi cel de-al doilea operand, pe nivelul de sub vârf): adică se elimină din stivă operatorul şi cel de-al doilea operand şi se pune în vârful stivei rezultatul operaţiei; dacă în vârful stivei nu se află un operator, operandul se adaugă în vârful stivei. Când se întâlneşte un operand şi în stivă se află o paranteză deschisă, operandul se pune în vârful stivei. La întâlnirea unei paranteze închise vom evalua toate operaţiile până la prima paranteză deschisă din stivă. Apoi vom înlocui paranteza deschisă cu rezultatul evaluării şi, dacă pe nivelul următor din stivă se gaseşte un operator, atunci efectuăm operaţia. La final, în vârful stivei vom avea rezultatul expresiei.

Problema 13-3. Matrice (Bursele Agora, 2000-2001)

Se consideră o matrice cu M linii şi N coloane ($1 \le M,N \le 100.000$). Fiecare element al matricii este *1*. Dorim să obţinem K ($0 \le K \le M \cdot N$) elemente egale cu *-1*, folosind o secvenţă de mutări de tipul următor: putem schimba semnul tuturor elementelor de pe o linie sau de pe o coloană.

Soluţie: Se observă că pe orice linie şi orice coloană, semnul elementelor se schimbă maxim o dată. De asemenea, nu contează care sunt liniile sau coloanele asupra căror aplicăm operaţia menţionată, astfel că, dacă aplicăm operaţia asupra a P linii (Q coloane), putem presupune că acestea sunt primele P linii (primele Q coloane). În plus, nu contază nici ordinea în care efectuăm operaţiile. Să presupunem că vrem să aplicăm operaţia asupra a P linii. După aplicarea celor P operaţii avem $P \cdot N$ elemente egale cu *-1* şi $(M-P) \cdot N$ elemente egale cu *1*. Dacă aplicăm apoi operaţia asupra a Q coloane, vom obţine $P \cdot N-P \cdot Q+(M-P) \cdot Q$. Aşadar, trebuie să găsim *2* valori întregi P şi Q ($0 \le P \le M$ şi $0 \le Q \le N$), astfel încât $P \cdot N+M \cdot Q-2 \cdot P \cdot Q=K$.

Vom încerca toate valorile posibile ale lui P. Pentru fiecare valoare, vom determina în timp $O(1)$ dacă există o valoare potrivită a lui Q. Pentru un P fixat, avem $Q=(K-P \cdot N)/(M-$

2·P). Dacă *(K-P·N)/(M-2·P)* este un număr întreg din intervalul *[0,N]*, atunci am găsit o valoare potrivită pentru *Q*.

Problema 13-4. Resturi 1 (Happy Coding 2005, infoarena)

Se dau *N* (*1≤N≤30*) numere (nu neapărat prime) distincte *p(1), p(2), ..., p(N)* (*2≤p(i)≤1.000*, *1≤i≤N*) şi *N* resturi distincte *r(1), r(2), ..., r(N)* (*0≤r(i)≤p(i)-1*; *1≤i≤N*). Aflaţi cel mai mic număr nenegativ *X* cu proprietatea: *X mod p(k) = r(k)*, pentru orice *k* între *1* şi *N*.

Exemplu:

N=3 **p(1)=5 ; r(1)=4** **p(2)=11 ; r(2)=3** **p(3)=19 ; r(3)=8**	**X=179.**

Soluţie: Această problemă este cunoscută sub numele de *Teorema Chineză a Restului*. Vom calcula numărul *X* cerut în mod incremental. La pasul *i* (*1≤i≤N*) vom avea calculat numărul *Q*, reprezentând cel mai mic număr care respectă primele *i* relaţii (deci *Q mod p(j)=r(j)* pentru orice *1≤j≤i*). La pasul *1*, *Q* este egal chiar cu *r(1)*. Având calculat numarul *Q* la pasul *i*, va trebui să determinăm numărul *Q'* la pasul *i+1* care respectă primele *i+1* relaţii. Acest număr *Q'* este de forma *Q+x·M*, cu *x≥0*, unde *M* este cel mai mic multiplu comun al numerelor *p(1), ..., p(i)* (*M* este egal cu produsul *p(1)·...·p(i)*, împărţit la cel mai mare divizor comun al numerelor *p(1), ..., p(i)*; *cmmdc(p(1), ..., p(i))=cmmdc(cmmdc(p(1), ..., p(i-1)), p(i))*). Ideea din spatele acestei formule este că la numărul *Q* trebuie adunat un multiplu al numărului *M*, pentru ca numărul obţinut *Q'* să păstreze aceleaşi resturi la împărţirile la primele *i* numere prime date. Avem acum următoarea ecuaţie: *Q+x·M=r(i+1)* (*mod p(i+1)*). Trecând pe *Q* în partea dreaptă, obţinem o ecuaţie de forma *A·x=B* (*mod P*), care se poate rezolva direct, folosind algoritmul lui Euclid extins, pentru a calcula inversul multiplicativ al lui *A*, relativ la numărul prim *P*. O metodă mai simplă de rezolvare a ecuaţiei (şi utilizabilă chiar şi atunci când nu există inverse modulare la fiecare pas, deoarece numerele *A* şi *P* din forma generică a ecuaţiei nu sunt prime între ele) se bazează pe a încerca toate valorile posibile pentru *x*, între *0* şi *p(i+1)-1*, care funcţionează deoarece valorile numerelor *p(*)* sunt relativ mici.

Problema 13-5. Resturi 2 (Lotul Naţional de Informatică, România, 2004)

Se dă un număr natural *N* (*1≤N≤10*) şi numerele naturale *p(1), p(2), …, p(N), r(1), r(2), …, r(N)*, unde *p(1), …, p(N)* sunt numere prime diferite două câte două şi *0≤r(i)≤p(i)-1*, pentru orice *i=1,...,N*. Spunem că un număr *X* este liber de resturi, dacă restul împărţirii lui *X* la *p(i)* este diferit de *r(i)*, pentru orice *i=1,...,N*. Considerăm şirul sortat al numerelor naturale libere de resturi. Să se determine al *K*-lea (*1≤K≤2.000.000.000*) element al şirului. Se garantează că rezultatul va fi mai mic decât 10^{10}.

Soluţie: Vom căuta binar cel mai mic număr *X* cu proprietatea că există exact *K* numere libere de resturi în intervalul de numere naturale *[0,...,X]*. Numărul găsit este răspunsul problemei. Mai trebuie doar să determinăm un algoritm eficient pentru a număra câte numere libere de resturi *NLR(X)* există într-un interval *[0,...,X]*. Vom folosi principiul includerii şi excluderii. Vom nota prin *NR(S,X)*=numărul de numere din intervalul *[0,...,X]* care, împărţite la fiecare număr *p(i)* din submulţimea *S* dau restul *r(i)*. Vom nota prin *NTR(h,X)* suma valorilor *NR(S,X)*, cu |*S*|=*h*. Avem *NLR(X)=(X+1)-NTR(1,X)+NTR(2,X)-NTR(3,X)+...+(-*

$1)^i \cdot NTR(i,X)+...+ (-1)^N \cdot NTR(N,X)$. Vom considera toate cele 2^N submulţimi *S* ale mulţimii *{p(1), ..., p(N)}* şi vom calcula *NR(S,X)* pentru fiecare. Vom folosi rezolvarea de la problema anterioară pentru a calcula cel mai mic număr *Q* care împărţit la fiecare număr *p(i)* din *S* dă restul *r(i)*. Dacă *Q>X*, atunci *NR(S,X)=0*. Altfel, vom calcula *M*=cel mai mic multiplu comun al numerelor *p(i)* din *S* şi vom avea *NR(S,X)=1+((X-Q) div M)* (prin *A div B* am notat câtul împărţirii întregi a lui *A la B*). Un caz particular apare când *r(i)=0* pentru toate numerele *p(i)* dintr-o submulţime *S*. În acest caz, *NR(S,X)=1+(X div M)*.

Algoritmul descris are complexitatea $O(2^N \cdot N \cdot log(X))$. Putem reduce complexitatea la $O(2^N \cdot log(X))$ dacă nu recalculăm de la început numărul *Q* corespunzător unei submulţimi *S*. Mai exact, să presupunem că am calculat numărul *Q'* corespunzător unei submulţimi *S'=S\{p(i)}* şi numărul *M'*=cel mai mic multiplu comun al elementelor din *S'*, unde *p(i)* este un element din *S*. Atunci *Q* se obţine imediat din *Q'* şi *M'*, considerând şi perechea *(p(i), r(i))* (avem ecuaţia $Q'+y \cdot M'=r(i)$ *(mod p(i))*; după ce determinăm cea mai mică valoare *y* care verifică ecuaţia, avem $Q= Q'+y \cdot M'$).

Dacă ne-ar interesa şirul sortat al numerelor care dau restul *r(i)* la cel puţin unul din numerele *p(i)* şi am dori să determinăm al *K*-lea număr din acest şir, am folosi o procedură asemănătoare. Vom căuta binar cel mai mic număr *X* pentru care există în intervalul *[0,X] K* numere care împărţite la cel puţin un număr *p(i)* dau resul *r(i)*. Avem nevoie şi de această dată de un algoritm eficient pentru a calcula *NN(X)*=numărul de numere din intervalul *[0,X]* care împărţite la cel puţin un număr *p(i)* dau resul *r(i)*. Observăm uşor că *NN(X)=(X+1)-NLR(X)*.

Problema 13-6. Transformări geometrice (SGU/.campion 2008)

Se dau *N* (*1≤N≤50.000*) puncte în spaţiul 3D. Asupra fiecărui punct vrem să efectuăm aceeaşi secvenţă de operaţii. Secvenţa constă din *M* (*1≤M≤50.000*) operaţii. Operaţia *i* (*1≤i≤M*) este de tipul *op(i)* (*Translaţie*, *Rotaţie în jurul unei drepte date din spaţiu* şi *Scalare*) şi se execută de *x(i)* ori consecutiv (*1≤x(i)≤1.000.000.000*). Determinaţi coordonatele finale ale celor *N* puncte.

Soluţie: Vom reprezenta fiecare operaţie sub forma unei matrici de transformare *4x4*. Fiecare punct va fi considerat ca având *4* coordonate, cea de-a *4*-a fiind mereu egală cu *1*. Matricile pentru translaţie şi scalare au forme standard. Matricea pentru o rotaţie în jurul unei drepte arbitrare se obţine după cum urmează:

(1) se translatează dreapta pentru a trece prin originea sistemului;
(2) se roteşte dreapta în jurul axei *OZ* până ajunge în planul *OXZ*;
(3) se roteşte dreapta în jurul axei *OY*, până se suprapune cu *OX*;
(4) se realizează rotaţia în jurul axei *OX*;
(5) se efectuează transformările *(3)*, *(2)* şi *(1)* (în această ordine) în sens invers.

O dată ce avem calculată matricea de transfromare *T(i)* pentru operaţia *i*, o vom ridica la puterea *x(i)*. Pentru aceasta, vom folosi ridicare la putere logaritmică. Vom calcula $T(i)^{x(i)}$ în *O(log(x(i)))* paşi, obţinând o matrice $T_2(i)$. Se calculează apoi matricea produs $TM=T_2(M) \cdot T_2(M-1) \cdot ... \cdot T_2(1)$. Pentru a determina coordonatele finale ale fiecărui punct *k* (*1≤k≤N*), înmulţim matricea *TM* cu vectorul coloană al coordonatelor punctului *k* (a *4*-a coordonată este *1*, după cum am menţionat deja). Vectorul coloană rezultat reprezintă coordonatele finale ale punctului *k*.

Complexitatea algoritmului este $O(M \cdot log(max(x(*)))+M+N)$.

Capitolul 14. Backtracking

Problema 14-1. Regine2 (Happy Coding 2007, infoarena)

Se dă o tablă de şah de dimensiune *NxN*. Pe această tablă, unele pătrăţele sunt libere, iar altele sunt blocate. Determinaţi care este numărul maxim de regine care pot fi plasate pe tabla de şah, astfel încât oricare două regine să nu se atace una pe alta. O regină poate fi plasată numai pe un pătrăţel liber. Două regine se atacă una pe alta dacă sunt pe aceeaşi linie, coloană sau diagonală şi toate pătrăţelele dintre cele *2* regine (de pe linia, coloana sau diagonala respectivă) sunt libere. Fie *Q* numărul maxim de regine care pot fi aşezate pe tablă. În plus, trebuie să determinaţi şi numărul de posibilităţi de a aşeza cele *Q* regine pe tablă.

Exemplu:

N=4 (. =liber ; **#**=ocupat) .#.. ##.. .#.# .#.#	Se pot amplasa maxim *4* regine, în *6* feluri.

Soluţie: Pentru fiecare poziţie (în ordinea liniilor şi, pentru fiecare linie, în ordinea coloanelor), se încearcă amplasarea sau neamplasarea unei regine în poziţia respectivă, iar apoi se marchează toate poziţiile atacate de regina respectivă, pentru a nu se mai încerca amplasarea unei regine viitoare pe o poziţie deja atacată. La întoarcerea din backtracking, poziţiile marcate se demarchează (vom folosi, de fapt, un contor pentru fiecare poziţie, în care vom reţine de câte regine deja amplasate este atacată poziţia respectivă). Singura optimizare necesară este că, atunci când marcăm poziţiile atacate de o regină nou-amplasată, vom marca doar poziţiile pe care vom încerca să amplasăm o regină în viitor (adică doar înspre direcţiile: est, sud-est, sud, sud-vest, şi nu în toate cele *8* directii).

Problema 14-2. Promo (Olimpiada Naţională de Informatică, România 2007)

Compania ONIx comercializează *N* produse. Pentru a creşte vânzările, compania a pus la dispoziţia clienţilor *M* oferte promoţionale. Fiecare ofertă constă din exact *2* produse diferite, care sunt vândute împreună la un preţ mai scăzut decât dacă ar fi vândute separat (de exemplu, suc şi apă minerală). Produsele sunt identificate prin numere de la *1* la *N*, iar ofertele promoţionale prin numere de la *1* la *M*. Deoarece şi-au schimbat de curând aplicaţia *software* ce gestionează baza de date a companiei, angajaţii nu s-au obişnuit cu noul sistem şi, din neatenţie, unul dintre aceştia a şters toate informaţiile despre produsele şi ofertele existente. Singurele informaţii rămase sunt cele ale departamentului de statistică, care foloseşte o bază de date proprie. Aceste informaţii sunt reprezentate prin numărul *M* de oferte şi de toate cele *K* perechi de oferte ce au un produs în comun (în mod evident, oricare *2* oferte pot avea cel mult un produs în comun). Folosind informaţiile departamentului de statistică, determinaţi numărul de produse şi cele *2* produse din cadrul fiecărei oferte.

Soluţie: Problema cere determinarea unui graf pe baza grafului muchiilor acestuia (unde *2* muchii sunt adiacente dacă au unul din cele *2* capete în comun). Vom împărţi graful celor *M* muchii în componente conexe. Pentru fiecare componentă conexă vom ordona muchiile ce fac parte din ea (noduri în graful muchiilor) astfel încât orice muchie în afara de prima să

aibă cel puţin o muchie adiacentă cu aceasta undeva înaintea ei (folosind o parcurgere *DF*, de exemplu). Vom determina, pe rând, nodurile ce reprezintă capetele fiecărei muchii din componenta conexă curentă. Primei muchii îi atribuim *2* noduri noi. În continuare, fiecare muchie are cel puţin o altă muchie adiacentă deja poziţionată, astfel că unul din cele *2* capete ale muchiei curente este unul din cele *2* capete ale vecinului ce se află înaintea acesteia în ordonarea pe care am realizat-o. Vom încerca fiecare din aceste *2* capete ca prim capăt al muchiei curente. Drept al doilea capăt vom testa ori un nod nou, neatribuit niciunei alte muchii, ori unul din cele *2* noduri ale unei alte muchii care are capetele deja fixate şi care este vecină cu muchia curentă. O analiză atentă ne va conduce la concluzia că există o singură modalitate de a atribui cele *2* capete muchiei curente în condiţiile în care muchiile dinaintea acesteia au capatele deja fixate. Există doar *2* excepţii posibile :

- când am ajuns la a treia muchie şi aceasta este adiacentă atât cu prima muchie, cât şi cu a doua: cele *3* muchii pot fi aşezate în "stea" sau în "triunghi"
- când aşezăm a *4*-a muchie (sau a *5*-a, după caz) poate exista situaţia în care muchiile fixate anterior sunt muchiile unui subgraf bipartit *2x2*: în acest caz, muchia curentă poate uni ori cele *2* muchii din partea stângă a subgrafului bipartit, ori cele două noduri din partea dreaptă

Cele *2* excepţii nu pot apărea simultan. În concluzie, pentru fiecare componentă conexă vom avea de încercat cel mult două variante. Complexitatea algoritmului este $O(M^2)$, chiar dacă putem să îl implementăm folosind backtracking.

Bibliografie

[Andreica-COMM2008] M. I. Andreica, „Optimal Scheduling of File Transfers with Divisible Sizes on Multiple Disjoint Paths”, Proceedings of the 7th IEEE Romania International Conference "Communications", pp. 155-158, 2008.

[Andreica-CTRQ2008] M. I. Andreica, N. Tapus, „Optimal Offline TCP Sender Buffer Management Strategy”, Proceedings of the 1st International Conference on Communication Theory, Reliability, and Quality of Service, pp. 41-46, 2008.

[Andreica-ISPDC2008] M. I. Andreica, N. Tapus, „Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling”, Proceedings of the 7th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 285-292, 2008.

[Andreica-MCBE2008] M. I. Andreica, E.-D. Tirsa, C. T. Andreica, R. Andreica, M. A. Ungureanu, “Optimal Geometric Partitions, Covers and K-Centers”, Proceedings of the 9th WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE), pp. 173-178, 2008.

[Bender-RMQLCA2000] M. A. Bender, M. Farach-Colton, “The LCA Problem Revisited”, Lecture Notes in Computer Science, vol. 1776, pp. 88-94, 2000.

[CLRS] T. H. Cormen, C. E. Leiserson, L. Rivest, C. Stein, „Introduction to Algorithms”, 3rd edition, MIT Press, 2009.

[Eppstein1992] D. Eppstein, M. Overmars, G. Rote, G. Woeginger, „Finding Minimum Area k-gons”, Discrete & Computational Geometry, vol. 7, pp. 45-58, 1992.

[PA3DCG] S. Ferguson, „Practical Algorithms for 3D Computer Graphics”, CRC Press, 2001.

[campion] http://campion.edu.ro/

[Codechef] http://www.codechef.com/

[Codeforces] http://codeforces.com/

[infoarena] http://www.infoarena.ro/

[Quine] http://www.nyx.net/~gthompso/quine.htm

[SGU] http://acm.sgu.ru/

[TIMUS] http://acm.timus.ru/

[TopCoder] http://www.topcoder.com/

[UVA] http://uva.onlinejudge.org/